Optimal Cycle Dating of Large Financial Time Series

by

Konrad Phillip Kapp

Submitted in fulfillment of the requirements for the degree of
Magister Scientiae
in the
Faculty of Science
at the
Nelson Mandela Metropolitan University

Supervisor: Prof. I.N. Litvine

2017
Declaration by candidate

Name: Konrad Phillip Kapp

Student number: 210076453

Qualification: MSc Mathematical Statistics

Declaration:

In accordance with Rule G4.6.3, I hereby declare that this dissertation is my own work and that it has not previously been submitted for assessment to another university or for another qualification.

Signature: .............................................

Date: .............................................
Acknowledgements

I would like to thank

- my supervisor, Prof. I.N. Litvine, for his constant support, willingness to exchange ideas, and excellent counsel;
- Werner Kapp, for his highly valued input on the language in this dissertation;
- my parents and family, for their ever present support, both personal and financial;
- the staff and postgraduate students of the NMMU statistics department, for their encouragement and appreciation of my work.
Abstract

The study of cycles in the context of economic time series has been active for many decades, if not centuries; however, it was only in recent decades that more formal approaches for identifying cycles have been developed. Litvine and Bismans (2015) proposed a new approach for dating cycles in financial time series, for purposes of optimising buysell strategies. In this approach, cycle dating is presented as an optimisation problem. They also introduced a method for optimising this problem, known as the hierarchical method (using full evaluation 2, or HR-FE2). However, this method may be impractical for large data sets as it may require unacceptably long computation time.

In this study, new procedures that date cycles using the approach proposed by Litvine and Bismans (2015), were introduced, and were specifically developed to be feasible for large time series data sets. These procedures are the stochastic generation and adaptation (SGA), buy-sell adapted Extrema importance identity sequence retrieval (BSA-EIISR) and buy-sell adapted bottom-up (BSA-BU) methods. An existing optimisation technique, known as particle swarm optimisation (PSO), was also employed. A statistical comparison was then made between these methods, including HR-FE2. This involved evaluating, on simulated data, the performance of the algorithms in terms of objective function value and computation time on different time series lengths, Hurst exponent, and number of buy-sell points. The S-Race methodology (T. Zhang, Georgiopoulos, and Anagnostopoulos 2013) was then applied to these results in order to determine the most efficient methods. It was determined that, statistically, SGA, BSA-EIISR and BSA-BU are the most efficient methods. Number of buy-sell points was found to have the largest effect on relative performance of these methods. In some cases, the Hurst exponent also has a small effect on relative performance.

**Keywords:** time series, Hurst exponent, cycles, optimisation, racing algorithms, efficiency, big data, computational statistics, swarm intelligence
Contents

List of tables ix
List of figures x
List of algorithms xii
List of abbreviations xiii
List of symbols xiv

1 Introduction 1
   1.1 Problem statement ........................................ 2
   1.2 Research aims and objectives ............................... 3
   1.3 Importance of research .................................... 4
   1.4 Literature review .......................................... 5
   1.5 Computational tools ........................................ 13
   1.6 Outline of paper ........................................... 13

2 Algorithms 15
   2.1 Profit objective function ................................. 15
2.2 Hierarchical method ............................................. 17
  2.2.1 Working example ........................................... 20
  2.2.2 Stopping conditions ....................................... 20
  2.2.3 Pseudocode ................................................. 20
2.3 Stochastic generation and adaptation (SGA) ....................... 23
  2.3.1 Stopping conditions ....................................... 24
  2.3.2 Working example ........................................... 25
  2.3.3 Parameters .................................................. 25
  2.3.4 Pseudocode ................................................. 29
2.4 BSA-EIISR ....................................................... 31
  2.4.1 Finding optimal troughs: approaches ....................... 32
  2.4.2 Finding optimal troughs: the method ....................... 34
  2.4.3 Stopping conditions ....................................... 35
  2.4.4 Pseudocode ................................................. 38
2.5 BSA-BU .......................................................... 41
  2.5.1 Stopping conditions ....................................... 43
  2.5.2 Pseudocode ................................................. 43
  2.5.3 Working example ........................................... 44
2.6 Particle swarm optimisation (PSO) ............................... 45
  2.6.1 Constraint handling ....................................... 48
  2.6.2 Parameters .................................................. 50
  2.6.3 Stopping conditions ....................................... 56
  2.6.4 Initialisation of swarm ..................................... 56
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.5</td>
<td>Synchronous vs asynchronous updates</td>
<td>57</td>
</tr>
<tr>
<td>2.6.6</td>
<td>Working example</td>
<td>58</td>
</tr>
<tr>
<td>2.7</td>
<td>PSO-SGA</td>
<td>58</td>
</tr>
<tr>
<td>2.8</td>
<td>Summary</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>Research methodology</td>
<td>61</td>
</tr>
<tr>
<td>3.1</td>
<td>Optimisation problem: objective function</td>
<td>61</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Profit objective function</td>
<td>61</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Time objective function</td>
<td>62</td>
</tr>
<tr>
<td>3.2</td>
<td>Time series simulation</td>
<td>63</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Hurst exponent</td>
<td>63</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Continuous-time fractional Brownian motion</td>
<td>65</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Discrete-time fractional Brownian motion</td>
<td>66</td>
</tr>
<tr>
<td>3.2.4</td>
<td>The fractional Black-Scholes model</td>
<td>66</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Generation of fBm</td>
<td>72</td>
</tr>
<tr>
<td>3.3</td>
<td>S-Races</td>
<td>72</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Pareto dominance</td>
<td>73</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Statistical error</td>
<td>73</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Sign test</td>
<td>74</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Holm-Bonferroni method</td>
<td>75</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Dominance matrix</td>
<td>76</td>
</tr>
<tr>
<td>3.3.6</td>
<td>S-Race description and pseudocode</td>
<td>77</td>
</tr>
<tr>
<td>3.3.7</td>
<td>Type I error of S-Race algorithm</td>
<td>78</td>
</tr>
<tr>
<td>3.4</td>
<td>Investigation of inputs</td>
<td>81</td>
</tr>
</tbody>
</table>
3.4.1 Time series length ........................................ 81
3.4.2 Number of buy-sell points ................................. 81
3.4.3 Hurst exponent ............................................. 82
3.5 Summary ..................................................... 82

4 Results ......................................................... 83
4.1 Introduction .................................................. 83
  4.1.1 Algorithms ............................................... 85
  4.1.2 Baseline plots ............................................ 86
4.2 Deterministic methods ....................................... 88
  4.2.1 S-Race results: baseline plots and summary .......... 89
  4.2.2 S-Race results: discussion .............................. 94
4.3 HR-SGA ....................................................... 96
  4.3.1 S-Race results and summary ............................. 97
4.4 SGA ........................................................ 98
  4.4.1 S-Race results: baseline plots and summary .......... 99
  4.4.2 S-Race results: discussion .............................. 103
4.5 PSO parameter tuning ....................................... 106
  4.5.1 PSO computation time .................................. 106
  4.5.2 All parameters .......................................... 109
4.6 HR-PSO-SGA .................................................. 111
  4.6.1 S-Race results and discussion ........................... 112
4.7 PSO-SGA ....................................................... 113
  4.7.1 S-Race results: baseline plots and summary .......... 114
4.7.2 S-Race results: discussion ................................................. 118

4.8 PSO ................................................................. 118
  4.8.1 S-Race results: baseline plots .................................... 119
  4.8.2 S-Race results: discussion ........................................ 122

4.9 Summary ............................................................. 122
  4.9.1 Hurst exponent and fitness value ................................ 123

5 Conclusions and recommendations .................................. 124
  5.1 Efficient methods ..................................................... 124
  5.2 Recommendations ................................................... 124
    5.2.1 If computation time is important ............................. 125
    5.2.2 If profit fitness is important ................................ 125
    5.2.3 Different preferences between fitness and time .......... 125
    5.2.4 Configuration: number of buy-sell points ................. 125
    5.2.5 Configuration: Hurst exponent ............................... 126
    5.2.6 Configuration: series size ................................... 126
    5.2.7 Stochastic vs deterministic .................................. 126
    5.2.8 Robustness ...................................................... 127
  5.3 Topics for future research ......................................... 127

Bibliography ............................................................ 129

Appendices ............................................................... 134

A Data structures used for algorithms ............................ 135
# List of tables

4.1 Configurations that S-Race is initially run on (two-level factorial design) ........................................... 85

4.2 Final pool after S-Race for deterministic methods (two-level factorial design) ........................................... 92

4.3 Final pool after S-Race for EIISR and BU (Additional configurations for three-level factorial design, where \( K = 10 \)) ................................................................. 93

4.4 Rejection criteria in S-Race for BU, where \( K = 10 \), and assuming that EIISR and BU are the only models in S-Race ................................................................. 96

4.5 Final pool after S-Race for SGA (two-level factorial design) ................................................................. 102

4.6 All configurations where SGA(5) did not survive S-Race ...................................................................... 105

4.7 All configurations where SGA with high population sizes survived S-Race ........................................... 105

4.8 Maximum possible computation times for PSO, denoted \( T_{\text{upper}}(|\Theta|) \), where \( K \geq 10 \) ................................................................. 107

4.9 Maximum possible computation times for PSO, where \( K = 2 \) ............................................................. 108

4.10 Distributions for each parameter ............................................................................................................ 111

4.11 Average fitness values for BU, EIISR and SGA(5) in S-Race where \( Y_{\text{size}} = 32000 \) and \( K = 10 \) ................................................................. 123

B.1 Performance of each model in S-Race appendix example ............................................................................. 147

B.2 Actual minimum confidence levels for S-Races run in this study .............................................................. 148

B.3 Number of prominent troughs in each window size ................................................................................. 149
# List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Cycle dating example</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Cycle dating example with BB algorithm</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Illustration of piecewise linear representation</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Hierarchical method working example</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>SGA working example</td>
<td>26</td>
</tr>
<tr>
<td>2.3</td>
<td>Illustration of how sub-problem two is solved in BSA-EIISR</td>
<td>33</td>
</tr>
<tr>
<td>2.4</td>
<td>BSA-EIISR clearance set illustration</td>
<td>36</td>
</tr>
<tr>
<td>2.5</td>
<td>BSA-BU working example</td>
<td>44</td>
</tr>
<tr>
<td>2.6</td>
<td>PSO neighbourhood types</td>
<td>54</td>
</tr>
<tr>
<td>2.7</td>
<td>Influence of Hurst exponent on fitness landscape</td>
<td>55</td>
</tr>
<tr>
<td>2.8</td>
<td>PSO working example</td>
<td>59</td>
</tr>
<tr>
<td>3.1</td>
<td>Illustration of the effects of the Hurst exponent</td>
<td>64</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of a baseline plot, with EIISR as the baseline method</td>
<td>87</td>
</tr>
<tr>
<td>4.2</td>
<td>Baseline plots for S-Race with all deterministic methods, where $K = 10$</td>
<td>89</td>
</tr>
<tr>
<td>4.3</td>
<td>Baseline plot for S-Race with all deterministic methods, where $K = 50$</td>
<td>90</td>
</tr>
<tr>
<td>4.4</td>
<td>Baseline plots for S-Race with all deterministic methods, where $K = 100$</td>
<td>91</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.5 More detailed baseline plot for EIISR and BU methods ........................................... 94
4.6 Baseline plot for S-Race with HR-SGA, where $K = 50$ ........................................... 98
4.7 Baseline plots for S-Race with SGA, where $K = 10$ ............................................... 99
4.8 Baseline plot for S-Race with SGA, where $K = 50$ ............................................... 100
4.9 Baseline plots for S-Race with SGA, where $K = 100$ .......................................... 101
4.10 Baseline plot example for S-Race with PSO-SGA, where $K = 2$ ....................... 113
4.11 Baseline plots for S-Race with PSO-SGA, where $K = 10$ .................................. 115
4.12 Baseline plot for S-Race with PSO-SGA, where $K = 50$ .................................... 116
4.13 Baseline plots for S-Race with PSO-SGA, where $K = 100$ .................................. 117
4.14 Baseline plots for S-Race with PSO, where $K = 10$ ............................................. 119
4.15 Baseline plot for S-Race with PSO, where $K = 50$ ............................................. 120
4.16 Baseline plots for S-Race with PSO, where $K = 100$ ........................................... 121

A.1 Representation of a dynamic array containing nine elements .................................. 135
A.2 Representation of a singly linked list ................................................................. 136
A.3 Illustration of removal from a linked list ................................................................... 137
A.4 Illustration of removal from a linked list ................................................................... 137
A.5 Illustration of insertion in a linked list ...................................................................... 138
A.6 Representation of a doubly linked list ................................................................. 138
A.7 Removal in a doubly linked list .............................................................................. 139
A.8 Representation of a min-heap ................................................................................. 139
A.9 Six buy-sell points during execution of BSA-BU .................................................... 143
A.10 Representation of buy-sell points using a doubly linked list .................................. 143
A.11 Illustration of how the min-heap and linked list were used together in BSA-BU .... 144
B.1 Illustration of how number of troughs that are lowest in different window sizes varies ......................................................... 150
B.2 SGA: four buy-sell points randomly generated .......................... 152
B.3 Illustration of how the solution is obtained if the first point is adjusted first. 153
B.4 Illustration of how the solution is obtained if the second point is adjusted first 153
B.5 Illustration of the normal case, where neighbourhoods do not go over the “edge” of the numbering of particles .............................. 154
B.6 Illustration of the case where the leftmost index of the neighbourhood is \( n_l = P_{PSO} + n_i - \frac{n}{2} \) ...................................................... 155
B.7 Illustration of the case where the rightmost index of the neighbourhood is \( n_r = n_i + \frac{n}{2} - P_{PSO} \) ........................................ 155
C.1 PSO model parameters where \( K = 2 \) ................................. 158
C.2 PSO model parameters where \( K \geq 10 \) ............................... 159
D.1 Baseline plots for S-Race with HR-SGA, where \( K = 10 \) .............. 161
D.2 Baseline plot for S-Race with HR-SGA, where \( K = 50 \) .............. 162
D.3 Baseline plots for S-Race with HR-SGA, where \( K = 100 \) ............ 163
D.4 Baseline plots for S-Race with PSO-SGA, where \( K = 2 \) and time series sizes are large ...................................................... 164
D.5 Baseline plot for S-Race with PSO-SGA, where \( K = 2 \) and time series sizes are medium ...................................................... 165
D.6 Baseline plots for S-Race with PSO-SGA, where \( K = 2 \) and series sizes are small ...................................................... 166
E.1 Example of raw data for S-Race with deterministic methods ........... 168
E.2 Example of raw data for S-Race with PSO ................................. 170
List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hierarchical method</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Core SGA algorithm</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>SGA algorithm</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>BSA-EIISR main procedure</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>BSA-EIISR procedure for extracting optimal troughs</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>BSA-BU algorithm</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>Generic PSO algorithm</td>
<td>46</td>
</tr>
<tr>
<td>8</td>
<td>Holm-Bonferroni procedure</td>
<td>76</td>
</tr>
<tr>
<td>9</td>
<td>S-Race algorithm</td>
<td>79</td>
</tr>
</tbody>
</table>
List of abbreviations

BSA - Buy-sell adapted
BU - Bottom-up
DE - Differential evolution
EIISR - Extrema importance identity sequence retrieval
fBm - Fractional Brownian motion
FE - Full evaluation
FE2 - Full evaluation 2 (Full evaluation for two buy-sell points)
FWER - Family-wise error rate
GA - Genetic algorithm
HR - Hierarchical
PLR - Piecewise linear representation
PSO - Particle swarm optimisation
SGA - Stochastic generation and adaptation
SI - Swarm intelligence
SSE - Sum of squared errors
List of symbols

$x_i$ - Time index of the $i^{th}$ observation in a given time series. Indices are ordered w.r.t. $i$, i.e.
if $i < j$, then $x_i < x_j$

$y(x_i)$ - Numerical value (on the y-axis) of the time series observation at time index $x_i$ of a
given time series

$Y_{x_i}$ - Same as the above

$Y_{size}$ - Size of a given time series

$N^*$ - Number of local peaks and troughs in a given time series

$\hat{x}_i$ - The $i^{th}$ peak/trough in a given time series

$K$ - Number of buy-sell points that are extracted from a time series

$X^*$ - Set of buy-sell points

$x_i^*$ - The $i^{th}$ buy-sell point in set $X^*$

$C$ - A set consisting of algorithms/methods

$C_i$ - The $i^{th}$ algorithm in set $C$

$C_{ij}$ - The $j^{th}$ execution of the $i^{th}$ algorithm in set $C$

$f_{profit}(X^*)$ - Profit objective function, evaluated on buy-sell points $X^*$

$g_{profit}(C_{ij})$ - Profit objective function for execution $C_{ij}$

$g_{time}(C_{ij})$ - Time objective function for execution $C_{ij}$

$\Theta$ - Domain of the objective function
\( B_t^H \) - Observation at time \( t \) of a fractional Brownian motion process

\( H \) - Hurst exponent

\( Q \) - Adjustment queue (used in SGA method)

\( V_i \) - Clearance set at iteration \( i \) (used in BSA-EIISR method)

\( U(a,b) \) - Uniform distribution with range \([a,b)\)

\( N(\mu,\sigma) \) - Normal distribution with mean \( \mu \) and standard deviation \( \sigma \)

\( Binom(n,p) \) - Binomial distribution with \( n \) trials and probability of success \( p \)

\( |A| \) - The cardinality of a set \( A \) (i.e. the number of elements in \( A \)), unless stated otherwise

\( A \setminus B \) - Set-theoretic difference of sets \( A \) and \( B \) (i.e. all elements in \( A \) that are not in \( B \))

\( A \cup B \) - Union between two sets \( A \) and \( B \) (i.e. all elements that are either in \( A \) or \( B \))

\( Z \leftarrow Y \) - Denotes assignment. Here, the variable \( Z \) gets assigned the value of variable \( Y \)
Chapter 1

Introduction

The Merriam-Webster dictionary defines the word “cycle” as “a repeating series of events or actions”. The keyword here is “repeating”: this implies that there is some regularity to these events, which in turn implies that inference may be made with regards to future events with some degree of confidence.

The study of cycles is an activity that mankind has been interested in for a very long time. This is not surprising, since cycles abound in nature: in astronomy, there are orbital, solar and carbon-nitrogen-oxygen (CNO) cycles; in physics, light and sound are studied as waves, which are almost synonymous with cycles; in agriculture, the cycles of seasons have long been of great importance to farmers.

There are many fields and professions where predictive power with regard to future events is important, such as medicine, engineering and city planning, for example. In economics and finance, policymakers and portfolio managers can make better decisions when informed of highly probable events that might occur in the future. This is one of the reasons why business cycles have been of interest to economists for nearly two centuries - business cycles often occur with surprising regularity, allowing economists to anticipate and act on these regular occurrences in the economy.

When one attempts to discover cycles in a time series, one generally wants to find segments where the time series has an overall upward trend, which is followed by a segment where the time series has an overall downward trend. This is equivalent to finding alternating peaks and troughs that separate the upward and downward trend segments. The process
1.1 Problem statement

While much formal research has been done on identifying and dating cycles in the context of economic time series, little has been done with regards to financial time series, such as series of asset prices. Furthermore, few, if any, methods exist for revealing cycles in financial data with a high sampling frequency (e.g. a sampling interval of less than a day), such as intraday asset price data.

Most of the existing methods (or alternatively, algorithms, techniques - these terms will be used interchangeably) for identifying cycles in economic or financial time series also lack an objective measure of the effectiveness of the identification of cycles in a particular series. Litvine and Bismans (2015) proposed a class of methods, known as the hierarchical methods, that find cycles by optimising an objective function. Using an objective function makes it possible to compare methods with less ambiguity, using a numerical quantity to measure how
1.2 RESEARCH AIMS AND OBJECTIVES

effective the identification of cycles is, or how “good” a method performs on a particular series. This allows a statistical analysis for comparison of different methods and time series to be done much easier.

While the hierarchical class of methods are designed to optimise a group of specified objective functions for the purposes of dating cycles, little is known about whether this class of methods really finds an optimal solution to the objective function, and whether alternative methods may find improved solutions. Furthermore, it is not known whether other classes of methods may require less time to obtain the same solution in terms of computation time taken by a computer program.

Time to obtain a solution is crucial when considering that some practitioners may wish to apply these methods to high sampling frequency time series of asset prices. These time series often contain tens of thousands of observations. Consider per-second data of share prices for a single trading day: such a time series may contain at least 27 000 observations if share prices are sampled at a period of a second or less. Applying methods that are inefficient may therefore prevent practitioners and researchers from doing analysis on these time series data sets in reasonable time.

1.2 Research aims and objectives

Since the hierarchical class of methods, proposed by Litvine and Bismans (2015), is the only known methodology that uses an objective function in the dating procedure, we aim to introduce new algorithms that might be considered alternatives to the hierarchical methodology, or that improve on the existing hierarchical methods. The improvements we desire are mainly higher values obtained for the specified objective function, and/or lower computation times. We also aim to present an analysis of the strengths and weaknesses of each method, as well as recommendations on the usage of each.

The aims above will be achieved by

- developing new and adapting existing algorithms for the purpose of optimising the desired objective function;

- selecting the methods that hold the most promise through preliminary testing. This
1.3. IMPORTANCE OF RESEARCH

is done via visual inspection of the results (e.g. do the cycles that this method gives us seem reasonable?);

- selecting the algorithms that are not outperformed by any other method in terms of both fitness (a more desirable objective function value) and time required to finish computation/optimisation;

  - this will be achieved by performing statistical comparisons between methods via hypothesis testing. A method is rejected if it is outperformed by at least one other method with a desired confidence level;

  - the result of this stage of the research is a set of efficient algorithms that always have a trade-off between one another. For example, if we switch from algorithm A to algorithm B, then either objective function value will be better, or time required for computation will be less, but not both or neither;

- investigating the differences between each of the efficient algorithms to obtain insight into their relative performance with varying time series sizes, Hurst exponents and number of buy-sell points. Note that since we wish to compare the methods as objectively as possible, it is important that each of the methods provide a means to strictly control how many buy-sell points are extracted, since number of buy-sell points directly influence objective function value (this will become clear in the next chapter).

1.3 Importance of research

The dating procedure can be seen only as a preprocessing step, but there are various possible ways in which the preprocessed data can then be used for further statistical analysis. For example, the distribution of length (measured in time) and height (measured in currency) of financial cycles may be analysed to provide support for traders and risk managers in decision making. Another possibility is to investigate the synchronisation of cycles between various asset and/or commodity prices in the same fashion that the synchronisation of business and
financial cycles have been investigated in Terrones, Kose, and Claessens (2011). Since an optimal solution for the problem defined here can be interpreted as optimal times to buy and sell a given asset (taking into account the constraints), it could also be used as a means to benchmark traders’ historical performance.

Keogh et al. (2004, pp. 1 - 9) and Fink, Pratt, and Gandhi (2003) discuss time series segmentation methods for various purposes, such as compression (selecting the most important points by which a time series may be represented) and support of further analysis (for example, as a similarity measure between different time series). The method used in this study to find cycles in financial time series may also be seen as a compression method that extracts the most prominent observations of a time series, providing an alternative to existing methods.

Since the focus will also be on designing methods that are appropriate for larger time series, practitioners will be able to analyse larger volumes of data using these methods. This will provide more information for the analysis of cycles in reasonable time.

1.4 Literature review

Novák, Perfilieva, and Dvořák (2016, p. 210) state that there are two basic approaches to analysing and forecasting time series: the Box-Jenkins Methodology, which is based on representation of the time series by a combination of autoregressive and moving average components (Novák, Perfilieva, and Dvořák 2016, p. 210), and the decomposition model, which assumes that the time series is the sum or product of the following components (Hyndman and Athanasopoulos 2014, pp. 146 - 147):

- Trend
  - A long-term increase or decrease in the data. The trend does not have to be linear.

- Season
  - Indicates that the time series is influenced by seasonal factors (such as the month,
1.4. LITERATURE REVIEW

quarter, etc.), which usually consist of rises and falls of fixed period in the data.

- Cycle
  - Recurring rises and falls in the data not of fixed period.

- Irregular
  - Remaining effects on the data after trend, cycle and season component effects have been removed.

For the purpose of this research, all time series will be viewed from the perspective of the decomposition model. Also, an important distinction in this paper is made between seasons and cycles: seasons occur at more or less fixed intervals, while cycles do not necessarily occur at fixed intervals. In other words, the period of seasons is fixed, while the periods of cycles are often not fixed. In this study, we will concern ourselves mainly with the task of finding cycles, as opposed to trends or seasons, in a given time series.

Edwards, Biscarri, and Gracia (2003, p. 928) state that there are two main approaches to finding cycles in economic variables. The first is a parametric approach that assumes some underlying process that switches between two states: one state corresponding to expansions, the other to recessions/contractions. The second is a non-parametric methodology that takes only the observed data into account in the search for prominent features that indicate the existence of cycles, which is the approach followed in this study.

Burns and Mitchell (1946, pp. 56 - 58) introduced the idea of finding cycles in economic data by identifying periods of rises and falls in a given time series. Peaks and troughs that define the turning points are then found, a process that they termed “dating cycles”.

Bry and Boschan (1971, p. 21) formalised the approach proposed by Burns and Mitchell (1946) by introducing an algorithm that identifies potential turning points in time series data. This method was designed specifically for economic time series data, which typically has a sampling interval of a month or more. Since a lot of economic data is available only
1.4. LITERATURE REVIEW

Figure 1.2: Example of dating cycles in simulated data with BB algorithm, which yields a reasonable result here. However, with the BB algorithm, it is very difficult to control exactly how many buy-sell points (or cycles) are revealed in the series, making it hard to compare it to other methods using an objective function.

in quarterly intervals, Harding and Pagan (2002, pp. 368 - 369) adapted the BB algorithm and simplified it (Pagan 2013, p. 5) to apply it to quarterly data, after which it became known as the BBQ algorithm. James Engle improved the BBQ algorithm by providing a modification to it, known as the modified BBQ algorithm (MBBQ), which proved to be a more reliable version of the BBQ algorithm (Pagan 2013, p. 5).

The BB and BBQ (and MBBQ) algorithms were designed specifically for economic data, and for monthly and quarterly intervals between observations, respectively. In this study, the focus is on finding algorithms that could also be used effectively on high sampling frequency data, such as time series of asset prices, which typically has a sampling interval of much less than a month, and is usually much more volatile (as opposed to the typically smoothed data of economic time series). It is possible to apply the BB-type algorithms to data of arbitrary sampling frequency, ignoring the sampling interval (see figure 1.2), but these methods were not designed to optimise the objective function used in this work for finding cycles. This is because the BB-type algorithms use mostly horizontal windows of neighbouring observations in its criteria for identifying cycles, rather than vertical distance between observations, which has the largest effect on the objective function. For example, in the BB algorithm, the trough of a cycle at time series index $x_i$ must meet the following criteria
1.4. LITERATURE REVIEW

\[ y(x_i) \leq y(x_{i+j}), \text{ for } j = 1, 2, ..., n \]

where \( y(x_i) \) is the value of an economic indicator at time series index \( x_i \), and \( n \) can be set to any natural number.

Furthermore, it is very difficult to control the final amount of cycles that the BB algorithm will identify. Therefore, if a specific number of buy-sell points are required, the BB method is even less desirable to use for optimisation of the objective function.

There has been a relatively recent surge of interest from the field of computer and information science in selecting the most important points from a given large and highly volatile time series data set (such as a high sampling frequency financial time series), as well as dividing the time series into segments that describe it best. These data reduction methods are mainly used for compression and indexing of time series, and as a data preprocessing task for forecasting and classification models, although more applications may exist. Unlike the BB class of methods, which assume low sampling frequency economic data, most of these methods make no assumption on the sampling frequency (such as Keogh et al. (2004) and Fink, Pratt, and Gandhi (2003)), while some others were designed specifically for higher sampling frequency financial time series (such as Wu and Huang (2009)). These methods were thought to hold promise in optimising the objective function, and are discussed next, after which stochastic optimisation algorithms will be discussed.

Keogh et al. (2004, pp. 1 - 9), provide descriptions of algorithms that obtain a representation of a time series known as a piecewise linear representation (PLR). A PLR divides the time series into linear segments that approximate it best. An example is shown in figure 1.3.
1.4. LITERATURE REVIEW

(a) The green lines connecting the red dots above are the linear segments that represent the time series. The PLR consists of these green lines.

(b) Illustration of the effectiveness of the PLR representation of the series.

Figure 1.3: Illustration of piecewise linear representation

Three general approaches to obtaining the PLR (Keogh et al. 2004, p. 4) exist:

- **Sliding windows**
  - A segment is grown until some error of approximation for that segment is exceeded. After this a new segment starts. This is continued until the end of the time series is reached.

- **Bottom-up**
  - Starts with small segments, then joins adjacent segments into larger segments until some stopping criterion is met.

- **Top-down**
  - Starts with large segments, then divides segments recursively into smaller segments until some stopping criterion is met.
To find the PLR, the methods above generally find linear segments that minimise the sum of squared errors (SSE) of the approximation of the time series via the lines connecting the extracted points (green lines in figure 1.3a), which does not link to the objective function we wish to maximise. Even so, the bottom-up and top-down methods are closely linked to two methods presented in this study. This is because to obtain the PLR, only the points on the edges of the segments are found and stored (the points shown as red dots in figure 1.3a), and these points are then used as a possible solution to the optimisation problem. With these methods, it is also much easier to control the number of buy-sell points extracted by setting the stopping criterion. More details will be given on these modified methods in chapter 2.

Wu and Huang (2009, pp. 501 - 504) proposed an algorithm that represents a time series with peaks and troughs that have a certain prominence in the series, using a similar approach to the BB-type algorithms. In this method, more prominent peaks and troughs are found in a sequential nature, which makes it easier to specify a set of rules for when the number of peaks and troughs extracted is enough, overcoming one of the shortcomings of the BB algorithm when applied to the optimisation problem assumed in this study. This made adaptation of this algorithm relatively easy. A more detailed discussion of the adapted method is given in chapter 2.

Since the dating procedure is achieved by optimising an objective function, drawing from the vast literature on optimisation methods was an obvious choice. However, since the objective function’s domain consists of discrete time series observations, and is not continuous in nature, many traditional optimisation methods could not be applied, since those usually rely on some continuity and differentiability to work. Stochastic optimisation algorithms are methods that either optimise objective functions that contain random noise, or that search for optimal solutions using random variables (Spall 2004, p. 172). The second requirement above refers to algorithms that make random perturbations to the current solution in order to improve the solution. This is done many times, until an optimal or near-optimal solution is found. One of the main advantages of these methods is that they do not require any continuity or differentiability in the objective function.

Genetic algorithms (GAs), which are a subset of stochastic optimisation algorithms, have already been used as a time series segmentation method in a few cases. Gacek and Pedrycz (2003, pp. 1203 - 1208) used a genetic algorithm to segment ECG time series data; Tseng
et al. (2006, pp. 443 - 447) also used a genetic algorithm, but combined it with a clustering and wavelet transform method to segment the time series into distinct patterns. Graves and Pedrycz (2009) used differential evolution, a type of GA, as a segmentation method for multivariate time series.

In this study, two stochastic optimisation algorithms were considered: differential evolution (DE) and particle swarm optimisation (PSO). PSO is a stochastic optimisation procedure introduced by Eberhart and Kennedy (1995, pp. 111 - 120). PSO does not fall under the paradigm of GAs, but rather under swarm intelligence (SI). In preliminary testing, PSO was found to give improved performance over DE for our purposes, therefore to limit the scope of this study, only PSO was considered for a more formal analysis. Note that PSO is a method that normally gets applied to optimisation problems where the domain is continuous. Even though the domain that PSO is applied to here is discrete, the time series that defines the domain here is normally very large. For this reason, it was thought that the domain may be treated more like a continuous space, which is why PSO was thought to be an appropriate method. More details on PSO are given in chapter 2.

As already stated, the hierarchical class of methods proposed by Litvine and Bismans (2015) was designed to optimise the required objective function for the purpose of finding cycles in data. A description of the hierarchical methodology is given in chapter 2. Note that this method follows the same approach as the top-down method discussed earlier, although it is slightly different in that it works by attempting to find vertical distances between buy-sell pairs, instead of constructing linear segments to minimise SSE.

Since there are many different algorithms that were compared in this study, a consistent method to compare them was needed. Since there are two criteria with which a method may be evaluated, some technique for dealing with multi-criteria comparisons had to be used. Three approaches were considered

- to combine both criteria into a single equation, so that a single value may be used to evaluate a given method. All methods are then evaluated on this single criteria. The downside to this approach, is that one needs to determine an appropriate equation for this purpose, which could require an in-depth analysis of its own.
- to keep one criteria constant (in this case, computation time) while evaluating all methods on the other criteria.
• to compare methods on both criteria simultaneously, which leads to an “efficient fron-
tier” of methods, which are not outperformed in one criteria, keeping another constant,
by any other method.

In this study, the third approach was followed. This approach facilitates the most thor-
ough study of the methods, since results on both criteria are easily assessed. For this ap-
proach, however, there exist, in turn, some options on how to implement it.

The simplest method for either multi- or single-criteria comparisons is a brute-force ap-
proach (Birattari et al. 2002, p. 13): run all algorithms on a certain number of problems
(time series, in this case), then select the methods that perform the best using some analysis
on all the data of their performance. The problems associated with this approach, are (1) one
needs to determine the appropriate parameter settings that should be tested (where applica-
able, such as with PSO), (2) the amount of sample problems necessary to run the algorithms
on to get a reliable estimate of performance of each method needs to be determined, and
(3) it could be computationally very expensive, since all methods are run an equal number
of times, regardless of their performance, thus computational resources are wasted on meth-
ods that were clearly inferior, and that did not need to be tested as extensively as other
well-performing algorithms (Birattari et al. 2002, p. 18).

To alleviate some of the problems described above, a family of model selection algorithms
known as racing algorithms may be employed. The main idea behind racing algorithms is
that a group of candidate models (known as a candidate pool) is tested repeatedly in parallel.
As soon as enough statistical evidence of inferiority has been accumulated for any of the
methods in the candidate pool, those methods get discarded without testing them further.
This saves significant computational resources, as reported by Yuan and Gallagher (2004, p.
180), Birattari et al. (2002, p. 18), and Maron and Moore (1997, p. 221).

The first to describe racing algorithms were Maron and Moore (1994, pp. 61 - 62) where
they used Hoeffding’s bound to distinguish between better and worse prediction models by
determining a confidence interval for each model’s prediction error. Birattari et al. (2002,
p. 11) were the first to extend this work to evolutionary algorithms (to which some of
the algorithms in this paper are related) by using a particular procedure known as the F-
Race. F-Race makes use of the Friedman test, a non-parametric statistical test developed
by Friedman (1937, p. 676) to detect differences in multiple test samples via rankings. It
is also possible to use an Analysis of Variance (ANOVA) test instead of Friedman test, in
which case the procedure will be known as A-Race.

The procedures described in the preceding paragraphs all assume there is a single measure of performance for each algorithm, unlike in this study, where there are two. A multi-objective racing algorithm is therefore needed. T. Zhang, Georgiopoulos, and Anagnostopoulos (2013, pp. 1566 - 1568) present a procedure known as S-Race, which is a racing algorithm that accommodates multi-objective evaluation of each model. S-Race also uses the sign test for matched pairs instead, where A-Race and F-Race use the ANOVA and Friedman tests, respectively. The sign test is a non-parametric procedure, as opposed to ANOVA in A-Race, so no requirements are made on the underlying data — a characteristic shared with the Friedman test. However, the presence of more than one criteria with which each method is evaluated may introduce some complications with regards to rankings, which is how the Friedman test differentiates between methods. The sign test, applied in a pairwise fashion on all models in the pool, is therefore an appropriate statistical procedure for use in the presence of more than one criteria. S-Race, the procedure used to determine the efficient algorithms, is described in more detail in chapter 3.

1.5 Computational tools

Two computational tools were used in this study: Wolfram Mathematica 10 and Python 3. Mathematica was used to simulate time series data, due to its excellent facility for generating fractional Brownian motion. Python 3 was used for implementation of all algorithms, with some supporting functionality from the Numpy, Matplotlib and Pandas libraries.

1.6 Outline of paper

The basic outline of the paper is as follows: Chapter 2 describes, in some detail, each cycle dating algorithm to be compared in this paper. This is done before any comparisons or comparison methodology is discussed, so that readers may be familiarised with what is being compared.

Chapter 3 deals with the research methodology followed. This includes descriptions of the stochastic processes used to simulate the data which the methods were applied to, as
well as some rationale behind why these specific processes were used. S-Race, the method used to compare algorithms’ performance, is also discussed in some detail.

Chapter 4 contains results and discussions, which include comparisons between the different algorithms. Results on performance of the algorithms with various types of time series and number of buy-sell points are given and discussed.

Chapter 5 contains conclusions and recommendations based on the findings in chapter 4. Ideas for future research are also discussed.
Chapter 2

Algorithms

In this chapter, detailed descriptions are given of the algorithms compared in this dissertation. This includes some of the algorithms described in the literature review as well as new methods proposed in this paper. Most of the methods and adaptations presented here are original, and introduced in this study. Care is therefore taken on the explanation of how they work, since there is no other source that the reader will be able to consult with regards to most of these algorithms. In the case of PSO, an existing algorithm, the focus is more on the rationale behind the way it was implemented in order to solve the optimisation problem introduced below.

2.1 Profit objective function

The algorithms presented in this chapter are all designed or adapted to optimise a particular objective function, variants of which most of these methods will also be able to optimise, with some small adjustments. This objective function is given in equation (2.1), and is the same as that which was referred to in the literature review.

We wish to solve the following problem

Maximise

over $x_i^*, i = 1, 2, ..., K$ (where K is even)
2.1. PROFIT OBJECTIVE FUNCTION

\[
f_{\text{profit}}(X^*) = \frac{K}{2} \sum_{i=1}^{K/2} (y(x_{2i}^*) - y(x_{2i-1}^*))
\] (2.1)

subject to

\[x_i^* < x_j^* \text{ if } i < j\]

where \(y(x)\) is the asset price at time \(x\) (or alternatively, time index \(x\)), and \(X^*\) is a set of the time indices of buy-sell points, i.e. \(X^* = \{x_1^*, x_2^*, \ldots, x_K^*\}\). Note that if we assume that buy-sell points are ordered according to their subscripts in the set \(X^*\), as shown above, then the constraint implies that the set must be ordered from smallest to largest. We will refer to the subscript of a buy-sell point (e.g. \(i\) in \(x_i^*\)) as its index in the set \(X^*\).

\(f_{\text{profit}}\) will be referred to as the profit objective function, profit fitness function, or simply fitness function. From equation (2.1), the profit objective function could alternatively be written as follows

\[
f_{\text{profit}}(X^*) = \frac{K}{2} \sum_{i=1}^{K/2} y(x_{2i}^*) - \frac{K}{2} \sum_{i=1}^{K/2} y(x_{2i-1}^*)
\] (2.2)

This representation makes it clear which components have a positive (first summation, sell points) and which have a negative (second summation, buy points) effect on the objective function.

**Definition 2.1 (Peak/trough).** A time series observation higher than at least its two neighbouring observations, will be referred to as a peak. On the other hand, an observation lower than at least its two neighbouring observations, will be denoted as a trough.

**Definition 2.2 (Optimal peak/trough).** Assume a final solution to the objective function is denoted \(X_{\text{final}}^*\). A time series observation that is part of the first (positive) summation in the representation of \(X_{\text{final}}^*\) given by equation (2.2), will be referred to as an optimal peak. A time series observation that is part of the second (negative) summation in equation (2.2) will be denoted as an optimal trough.

Note that an optimal peak/trough can also be seen as an optimal time to sell/buy. They may therefore also be referred to as sell or buy points. A buy-sell point refers to either a sell
or a buy point (i.e. no distinction is made).

The methods presented in this chapter all find approximate solutions to the optimisation problem specified above, so they are not guaranteed to find the true optimum to the objective function. Since the domain of \( f_{profit} \) is discrete, one way to find a guaranteed true optimum is to evaluate every possible solution \( X^* \). Such an approach will require vast computing power. Suppose we wish to find 10 optimal buy-sell points out of a time series of 5,000 observations. Assume that there are 2,000 peaks and troughs in this time series, and that only peaks and troughs are considered as possible buy-sell points. An algorithm that tests every combination will have to evaluate \( \binom{2000}{10} = 2.759 \times 10^{26} \) different solutions for \( f_{profit} \) to find the true optimum. This is for a time series of only 5,000 observations with 10 buy-sell points to be extracted. Later in this study it will become clear that this is a very short time series compared to other larger time series that the methods are also applied to. Ten buy-sell points are also a very low number compared to the higher quantities of buy-sell points that will also be extracted. If the size of the time series was doubled (assuming the number of peaks and troughs in the series also doubles) then there are \( \binom{4000}{10} = 2.857 \times 10^{29} \) different solutions that will be evaluated. Doubling the time series size therefore increased the number of solutions to be evaluated, and thus the time we have to wait for a solution, more than a thousandfold. Evaluating every possible solution is therefore impractical, especially as time series size increases, and thus methods that approximate a solution are required. Such methods are presented in this chapter.

### 2.2 Hierarchical method

The hierarchical method works by breaking the optimisation problem defined by equation (2.1) (profit fitness function) into smaller problems. In the first iteration, the first and last observations of the time series are assumed to be buy and sell points respectively. After this, in each iteration, a “best” segment of the time series is selected and a new buy-sell (buy point first) or sell-buy (sell point first) pair is inserted into this segment (thus, inserted into the set of buy-sell points \( X^* \)), where a segment is defined by two successive buy-sell points. In each iteration, the pair to be inserted into \( X^* \) is determined by optimising the following sub-problem

Maximise
over $x_{2i+1}^*, x_{2i+2}^*$

$$f_{long}(X_{long}) = y(x_{2i}^*) - y(x_{2i+1}^*) + y(x_{2i+2}^*) - y(x_{2i+3}^*) \quad (2.3)$$

subject to $x_i^* < x_j^*$ if $i < j$

if the buy-sell point defining the start of the segment, $x_{2i}^*$, is a sell point (which implies that the point defining the end of the segment, $x_{2i+3}^*$, is a buy point). Since $x_{2i}^*$ and $x_{2i+3}^*$ are given points, they are constant, so we have to find $x_{2i+1}^*$ and $x_{2i+2}^*$ that will maximise the above. We can therefore rewrite it as

Maximise

over $x_{2i+1}^*, x_{2i+2}^*$

$$f_{long}(X_{long}) = y(x_{2i+2}^*) - y(x_{2i+1}^*) \quad (2.4)$$

subject to $x_{2i}^* < x_{2i+1}^* < x_{2i+2}^* < x_{2i+3}^*$.

When the point defining the start of the segment is a buy point, we have

Maximise

over $x_{2i}^*, x_{2i+1}^*$

$$f_{short}(X_{short}) = -y(x_{2i-1}^*) + y(x_{2i}^*) - y(x_{2i+1}^*) + y(x_{2i+2}^*) \quad (2.5)$$

subject to $x_i < x_j$ if $i < j$

Once again, since this time $x_{2i-1}^*$ and $x_{2i+2}^*$ are constants (they define the start and end of the segment, respectively), the above reduces to

Maximise

over $x_{2i}^*, x_{2i+1}^*$

$$f_{short}(X_{short}) = y(x_{2i}^*) - y(x_{2i+1}^*) \quad (2.6)$$
subject to $x^*_{2i-1} < x^*_{2i} < x^*_{2i+1} < x^*_{2i+2}$.

The segment $S_L$ with the largest excess variation is selected as the “best” segment of the time series to perform the next optimisation step on. Excess variation is defined as follows

$$
\sum_{n=m+1}^{M} (|y(x_n) - y(x_{n-1})| - |y(x_M) - y(x_m)|)
$$

where $x_m$ and $x_M$ are the first and last observations in the segment, respectively. Excess variation is generally higher when there are more peaks and troughs in a time series. For this reason we choose the region with the highest excess variation, since this implies that there are more peaks and troughs and thus there exists a better chance of finding more optimal peaks/troughs.

The problems specified above are solved repeatedly until enough buy-sell points are in the set $X^*$. The method used to optimise the sub-problems in equations (2.4) and (2.6) may be specified by the user of the method. The simplest method to use for this purpose is one that simply checks every combination of peak and trough (if equation (2.6) is optimised) or every combination of trough and peak (if equation (2.4) is optimised). This method is referred to as the full evaluation 2 (FE2) method, and the hierarchical method with FE2 used in its optimisation steps is referred to as HR-FE2. Since the optimisation problem implied by equations (2.4) and (2.6) are special cases of the objective function in section 2.1, any of the other algorithms presented in this section can also be used in conjunction with the hierarchical method in this fashion. For example, when the hierarchical method is used with

- the stochastic generation and adaptation method (SGA), it is known as HR-SGA;
- the buy-sell adapted EIISR method, it is known as HR-EIISR;
- the buy-sell adapted bottom-up algorithm, it is denoted HR-BU;
- particle swarm optimisation (PSO), it is identified as HR-PSO.
2.2. HIERARCHICAL METHOD

2.2.1 Working example

A working example of the hierarchical method is shown in figure 2.1 on pages 21 and 22. Each row corresponds to an iteration: the left figure shows the current solution before the iteration is applied, and the figure on the right, the current solution after the iteration is finished.

2.2.2 Stopping conditions

The algorithm stops when the set $X^*$ has $K$ elements in it. Since $X^*$ initially contains two buy-sell points, and in each iteration two buy-sell points are added to $X^*$, the algorithm runs for $\frac{K}{2} - 1$ iterations.

2.2.3 Pseudocode

The hierarchical method in pseudocode form is given in algorithm 1 below.

<table>
<thead>
<tr>
<th>Algorithm 1 Hierarchical method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: procedure HIERARCHICAL($K, Y$)</td>
</tr>
<tr>
<td>2: $\hat{Y} \leftarrow \text{PreprocessedTimeSeries}(Y)$</td>
</tr>
<tr>
<td>3: $X^* \leftarrow {\hat{x}_1, \hat{x}_N^<em>}$  (\triangleright ) $X^</em>$ initially contains only the first and last points in $\hat{Y}$</td>
</tr>
<tr>
<td>4: while $</td>
</tr>
<tr>
<td>5:  Find $S_L$  (\triangleright ) Find segment with highest variation</td>
</tr>
<tr>
<td>6:  if $\hat{y}(x_m) &lt; \hat{y}(x_M)$ then  (\triangleright ) If segment starts with a trough and ends with a peak</td>
</tr>
<tr>
<td>7:  $f_{\text{obj}} \leftarrow f_{\text{short}}$</td>
</tr>
<tr>
<td>8:  else  (\triangleright ) If segment starts with a peak and ends with a trough</td>
</tr>
<tr>
<td>9:  $f_{\text{obj}} \leftarrow f_{\text{long}}$</td>
</tr>
<tr>
<td>10: end if</td>
</tr>
<tr>
<td>11: $x^<em>_i, x^</em><em>j \leftarrow \text{Optimise}(f</em>{\text{obj}} \text{ given } S_L)$  (\triangleright ) Find new positions by maximising $f_{\text{obj}}$</td>
</tr>
<tr>
<td>12: $X^* \leftarrow X^* \cup {x^<em>_i, x^</em>_j}$  (\triangleright ) Insert new positions into $X^*$</td>
</tr>
<tr>
<td>13: end while</td>
</tr>
<tr>
<td>14: end procedure</td>
</tr>
</tbody>
</table>

In line 2 in algorithm 1, the time series is preprocessed so that only peaks and troughs from the original time series remain. This is done since optimal buy-sell points will always
2.2. HIERARCHICAL METHOD

Figure 2.1: Hierarchical method working example

(a) First iteration: Whole time series is searched so that a sell, then a buy point may be obtained

(b) Result of first iteration: A sell then a buy point is found, since the point that defines the start of the region to be searched was a buy point

(c) Second iteration: The second region has the most excess variation out of all regions, therefore is chosen as the region to perform the next step on

(d) Result of second iteration: This time a buy, then a sell point is found, since the start of the region is defined by a sell point
2.2. HIERARCHICAL METHOD

(e) Third iteration: First region is found to be the best region for improvement of the solution

(f) Result of third iteration: sell, then a buy point is found for the same reasons stated in iteration one and two

(g) Final solution, superimposed over the original series

(h) Final solution, reduced series shown by itself
be on peaks/troughs, therefore having solutions that have buy-sell points that do not fall
on peaks or troughs in the domain, will waste computational resources. This preprocessing
step is done for all the algorithms, except the BSA-EIISR method presented in section 2.4.

2.3 Stochastic generation and adaptation (SGA)

A new method is now proposed, which will be referred to as the stochastic generation and
adaptation method (or SGA method). It is a procedure that can be considered to fall under
the umbrella of stochastic optimisation algorithms. Its details will be discussed here.

Assume that we wish to find \( K \) buy-sell points in the time series. Now, \( K \) points from
this time series are randomly, uniformly sampled without replacement (to avoid duplicates),
after which the indices of these \( K \) points are inserted into the set \( X^* \) (now with cardinality
\( K \)). Note that this is the same set \( X^* \) given in equation (2.1), where the \( i \)-th element of
\( X^* \) is given by \( x^*_i \) and \( x^*_i < x^*_j \) if \( i < j \). Each point in \( X^* \) is then adjusted according to
whether it should be a buy or sell point: if it is a sell point (which means every point at an
even index \( 2i \) of \( X^* \)), it is moved to the highest position between positions \( x^*_{2i-1} \) and \( x^*_{2i+1} \)
in the time series. This is because all sell points are part of the first (positive) summation in
equation (2.2), so the higher each of these points are in the time series, the higher the profit
fitness. Since each of the points have to keep their order (because of the constraint \( x_i < x_j \) if
\( i < j \)), the sell point is restricted to move to locations between its adjacent buy points. If a
buy point is adjusted (which means every point at an uneven index \( 2i-1 \) of \( X^* \)) it is moved
to the lowest point between positions \( x^*_{2i-2} \) and \( x^*_{2i} \) in the time series. This is because buy
points are all in the second (negative) summation in equation (2.2), so keeping the values
of these points as low as possible, while keeping their order, works toward maximising the
objective function.

From the above, it is easy to see that the SGA method also breaks the optimisation
problem given by the profit fitness function into smaller sub-problems. SGA effectively
takes each buy-sell point in the set \( X^* \) and maximises it, taking into account whether this
term is negative or positive in the objective function:

Maximise

over \( x^*_i \)
2.3. STOCHASTIC GENERATION AND ADAPTATION (SGA)

\[ f_{\text{sub}}(x_i) = \begin{cases} y(x^*_i) & \text{if } i \text{ is even} \\ -y(x^*_i) & \text{if } i \text{ is odd} \end{cases} \]  

subject to

\[ \begin{cases} x^*_{i-1} < x^*_i < x^*_i+1 & \text{if } 1 < i < K \\ 1 \leq x^*_i < x^*_{i+1} & \text{if } i = 1 \\ x^*_{i-1} < x^*_i \leq N^* & \text{if } i = K \end{cases} \]  

(2.7)

Therefore with the SGA method, maximising the objective function is attempted by maximising the \( K \) terms of equation (2.1) separately. This continues until none of the buy-sell points in \( X^* \) can be improved anymore. Note that the area between \( x^*_{i-1} \) and \( x^*_i \) will be referred to as accessible to buy-sell point \( x^*_i \), since that is the area where it is able to search for a better position. The area where \( x^*_i \) will search for an improved position, will be referred to as the searchable area (the distinction will become important soon). With this in mind, “none of the points can be improved anymore” means that all of the buy-sell points have searchable areas of size zero, which implies that no buy-sell points can improve their positions by searching again, thus no benefit in continuing the algorithm exists any longer.

2.3.1 Stopping conditions

Define

\[ Q = \{ x^*_{q_1}, x^*_{q_2}, x^*_{q_3}, \ldots, x^*_{q_p} \} \]  

(2.8)

where \( q_i \in \{1, 2, \ldots, n\} \). \( Q \) will be referred to as the adjustment queue, and describes the order in which buy-sell points will be adjusted, each index \( q_i \) referring to an index in the set \( X^* \). When the buy-sell point \( x^*_{q_i} \) is adjusted, it is taken out of the queue. We then have \( x^*_{q_2} \) at the front of the queue. New elements are added to the adjustment queue (always at the back) only when a buy-sell point \( x^*_{q_i} \) has its position changed. If \( x^*_i \) is adjusted to the right, then the accessible, and thus also the searchable area of \( x^*_{i-1} \) is enlarged if \( i > 1 \); if \( i = 1 \), then there are no points to the left of \( x^*_i \), therefore no buy-sell points’ searchable areas are enlarged by the point \( x^*_i \)’s move to the right. Therefore if \( i > 1 \), \( x^*_{i-1} \) is inserted into \( Q \) if it is not already in \( Q \). In the same way, if \( x^*_i \) is adjusted to the left, then the searchable area
of buy-sell point $x_{i+1}^*$ also gets larger, and must be inserted back into $Q$ if it is not already in $Q$. Once again, if it was $x_K^*$ (the rightmost buy-sell point) that was moved to the left, then the searchable areas of none of the other buy-sell points are enlarged, so no points are added to $Q$ in that case.

When $Q$ is empty, it means that none of the points in set $X^*$ can be improved anymore, which means there is no benefit in continuing the algorithm. The procedure is therefore stopped when $Q$ is empty. $X^*$ is then taken as the final solution.

2.3.2 Working example

An example of the procedure is given in figure 2.2 on pages 26 to 28. Each row of images corresponds to an iteration: the first image (in a row) shows the state of the solution at the start of the iteration. The second image shows the result of the iteration. Six iterations are shown.

As iteration five and six shows, the searchable area for a point is not always equal to its accessible area. This distinction is made so that a point will not search the same area twice, wasting computational resources. In iteration five, for example, the area from the second point to the left border of the yellow shaded region has already been searched for buy-sell point three. Buy-sell point three cannot improve its position by searching that area again, hence only the yellow shaded region, the searchable region, is searched. The distinction between searchable and accessible region can also be more formally defined in terms of set notation: let the set of time series observations in the accessible area for $pt_i$ be denoted by $A_i$, and the area that has already been searched for $pt_i$ by $B_i$. The searchable region, $E_i$, for point $i$, can then be defined as $E_i = A_i \setminus B_i$.

2.3.3 Parameters

Although it was not shown in preceding sections, the order of adjustment of buy-sell points in the SGA method has an influence on the final solution. This is shown in appendix B on page 152. The fact that the solution depends on the order in which observations are adjusted is where part of the name of the algorithm is derived from: it is obvious that the initial generation of buy-sell points is stochastic, but the adjustment of the points also
2.3. STOCHASTIC GENERATION AND ADAPTATION (SGA)

Figure 2.2: SGA working example

(a) Iteration one: second point is first in randomly chosen adjustment order. Point two therefore searches the highlighted yellow area for a better position.

(b) Iteration one result: second point is moved to highest point in searchable area and taken out of adjustment queue (therefore now indicated with black, instead of red).

(c) Iteration two: Point three is next. Since point three is a buy point, the lowest observation in the highlighted area is found and shown in the result to the right.

(d) Iteration two result. Note that the second buy-sell point is indicated in red again. This is because it is inserted back into the adjustment queue due to the third point’s movement to the right.
2.3. **STOCHASTIC GENERATION AND ADAPTATION (SGA)**

(e) Iteration three: The first point searches the highlighted yellow area for the lowest point, since it is a buy point.

(f) Iteration three result

(g) Iteration four: Point four, the last buy-sell point, searches the highlighted area for the highest point.

(h) Iteration 4 result. Point 3 is added to the back of the queue.
(i) Iteration five: The second point searches the indicated disjoint area, since the points either side of it moved away from it, both opening up new areas for point two. The rest of the region between point one and three has already been searched previously for point two.

(j) Iteration five result: No changes made to point two.

(k) Iteration six: The third point searches only the small indicated area since it is a newly opened area because of the fourth point’s movement to the right.

(l) Iteration six result: No changes made to point three. The adjustment queue is now empty, therefore this is the final solution.
introduces a stochastic element.

There are therefore two sources of randomness in the SGA method: the random generation of the initial buy-sell points (the initial solution, which gets adjusted), and the initial order in which those are adjusted. If just one set of random buy-sell points is generated, and is adjusted with a single initial order, then a greater chance exists that a “bad” set of initial points and a “bad” initial order of adjustment may yield an undesirable solution (a lower profit fitness).

What can also be done, is that a number of initial solutions may be generated. For each of these initial solutions, the algorithm could be run a number of times, each time with a different order of adjustment. This will each time yield a solution, and the best solution out of those may then be chosen as the final solution. This is expected to reduce the likelihood of obtaining a bad solution, as well as the variation in profit fitness from running the algorithm. This will be confirmed with respect to the number of initial solutions in chapter 4.

The number of initial solutions will be referred to as the population size (denoted $P_{SGA}$) and the number of times each individual (solution) in the population gets adjusted, with a unique order, as the individual iteration (denoted $I_{SGA}$). These two quantities will be the parameters for the full SGA method, shown in algorithm 3. Note that each time the population size is increased by $n$, the SGA method as described in algorithm 2 is run $n$ more times. Therefore population size should have a linear effect on the computation time taken by SGA (i.e. complexity with respect to population size is $O(n)$). In this study, SGA with population size $P_{SGA}$ is denoted SGA($P_{SGA}$).

The scope of this study is limited to investigating the effects of the population size only. Therefore, the individual iteration will be kept constant at one so that the population size may be investigated in more detail. Individual iteration may be studied in later research.

### 2.3.4 Pseudocode

The core SGA method is given in algorithm 2, with the full method in algorithm 3.
## 2.3. STOCHASTIC GENERATION AND ADAPTATION (SGA)

### Algorithm 2 Core SGA algorithm

1: procedure SGAcore(\(\hat{Y}, X^*\))
2: \(X^*\) is a randomly generated set of buy-sell points, passed to this procedure
3: \(Q ← \text{RandomPermutation}(X^*)\) \(\triangleright\) initialise the adjustment queue
4: while \(Q\) not empty do
5:   for each \(x_i^* \in Q\) do
6:     \(x_{\text{temp}} \leftarrow \text{FindBestPosition}(x_i^* \text{ given } \hat{Y})\)
7:     if \(x_{\text{temp}} > x_i^*\) then \(\triangleright\) if \(x_i^*\) moved to the right...
8:       if \(i > 1\) then \(\triangleright\) if there is a buy-sell point to the left of \(x_i^*\)...
9:       \(Q ← Q \cup \{x_{i-1}^*\}\) \(\triangleright\) add that point’s index in set \(X^*\) to the adjustment queue (if not already in queue)
10:   end if
11: else if \(x_{\text{temp}} < x_i^*\) then \(\triangleright\) if \(x_i^*\) moved to the left...
12:   if \(i < |X^*|\) then \(\triangleright\) if there is a buy-sell point to the right of \(x_i^*\)...
13:   \(Q ← Q \cup \{x_{i+1}^*\}\) \(\triangleright\) add that point to the adjustment queue (if not already in queue)
14: end if
15: end if
16: \(x_i^* \leftarrow x_{\text{temp}}\)
17: \(Q ← Q \setminus \{x_i^*\}\) \(\triangleright\) Since \(x_i^*\) was adjusted, take it out of queue
18: end for
19: end while
20: return \(X^*\)
21: end procedure
Algorithm 3 SGA algorithm

1: procedure SGA($P_{SGA}, I_{SGA}, K, Y$) 
2:     $\Phi \leftarrow \emptyset$ \hspace{1cm} \triangleright set of all solutions, empty initially 
3:     $\hat{Y} \leftarrow$ PreprocessTimeSeries($Y$) 
4: for $i$ from 1 to $P_{SGA}$ do 
5:     $X \leftarrow$ GenerateBSPoints($K$) 
6: for $j$ from 1 to $I_{SGA}$ do 
7:     $X^* \leftarrow$ SGAcore($\hat{Y}, X$) \triangleright Solution from SGA core, applied to same input, $X$ 
8:     $\Phi \leftarrow \Phi \cup \{X^*\}$ \hspace{1cm} \triangleright Add to existing set of solutions 
9: end for 
10: end for 
11: Return best solution out of $\Phi$ 
12: end procedure 

2.4 BSA-EIISR

In previous methods, in order to find the optimal buy-sell points, buy and sell points were generally found together. In the approach presented here, buy and sell points are found separately: first all the buy points, then all the sell points. Alternatively, all the sell points, then all the buy points may be found. To simplify this study, we will focus only on finding all buy points first, then sell points, but note that it may also be done the other way around.

We break the optimisation problem represented by the profit fitness function into two sub-problems. In the first sub-problem, the best troughs to use as buy points are found. This is the solution to the negative summation in equation (2.2):

Maximise

over $x^*_i$, for $i = 1, 2, ..., \frac{K}{2}$

$$f_{troughs}(X^*) = - \sum_{i=1}^{\frac{K}{2}} y(x^*_{2i-1})$$ (2.9)

subject to $x_i < x_j$ if $i < j$. 

Page 31
Once the optimal troughs have been found, all sell points are found by directly optimising
the positive summation in equation (2.2) by extracting the highest peaks between each pair
of buy points:

Maximise

over \( x^*_2i \), for \( i = 1, 2, ..., \frac{K}{2} \)

\[
 f_{\text{peaks}}(X^*) = \sum_{i=1}^{K/2} y(x^*_{2i}) \tag{2.10}
\]

subject to

\[
 \begin{cases}
 x^*_{2i-1} < x^*_{2i} < x^*_{2i+1} & \text{if } i < \frac{K}{2} \\
 x^*_{2i-1} < x^*_{2i} \leq Y_{\text{size}} & \text{if } i = \frac{K}{2}
\end{cases}
\]

where all \( x^*_j \), for \( j \) odd (the troughs found previously), are given constants.

The algorithm presented here to solve the optimisation problem, using this general ap-
proach, will mainly be concerned with optimising the first sub-problem (finding optimal
troughs), since the second sub-problem is fairly trivial to solve once the first has been solved.
Figure 2.3 demonstrates how sub-problem two is solved.

### 2.4.1 Finding optimal troughs: approaches

In order to find the optimal troughs, a fairly straightforward solution comes to mind: choose
the lowest \( \frac{K}{2} \) troughs in the time series. This directly maximises sub-problem 1, but what
usually happens in this case is that points from the lowest part of the time series are chosen,
often next to each other, or very close. This leaves very few options for choosing peaks
between each trough, since peaks and troughs should alternate. This usually leads to a
fairly low objective function value.

Another possible approach is to try every combination of \( \frac{K}{2} \) troughs, then choose the
highest peak between each pair of successive troughs, yielding a solution for each combination
of troughs. The solution that yields the highest objective function value is then selected as
the final solution. While this will lead to the optimal value of the optimisation problem, this
Figure 2.3: The points extracted in sub-problem two (black dots) are the highest points in each of the yellow shaded segments, demarcated by the vertical black lines. Each segment is defined by two successive troughs (red dots), found in sub-problem one, and the last segment by the last trough and the end of the time series. Sub-problem two is therefore fairly trivial to solve.
will most likely lead to extremely long computation times. Assume a time series is of size 10 000, for example, and that all the troughs in the time series are isolated (assume there are around 2000 troughs). Now, if 100 buy-sell points are to be extracted, 50 troughs are needed, therefore there are \( \binom{2000}{50} \approx 1.99 \times 10^{100} \) different solutions to evaluate, an infeasible amount.

Yet another approach is to choose the troughs that are the lowest in the largest windows around them in the time series (a quality which will be referred to from now on as prominence — a trough that is lowest in a larger window is more prominent than a trough that is lowest in a smaller window). This will solve the problem of not having enough observations between each pair of buy points for choosing a sell point, which was the problem in the first approach. Although, in this approach, troughs that are lowest in the time series are not directly chosen, it is a reasonable assumption that if troughs have a larger area around them, there are more peaks around them in those areas. There will therefore be more peaks to choose from for each optimal trough, which makes it more likely that higher peaks may be found for each trough, increasing the objective function value.

### 2.4.2 Finding optimal troughs: the method

The algorithm presented here is a combination of the second and third approaches. First, using the third approach, troughs that are less prominent are filtered out, which yields a much smaller set of troughs to choose from (the more prominent troughs). Since this set will always be very small (except in one case, which will be described shortly), the number of combinations of troughs tested will also be small, which brings the computational cost down to acceptable levels. Note that the third approach is based on the Extrema importance identity sequence retrieval (EIISR) algorithm invented by Wu and Huang (2009). The method introduced here is named after EIISR because its main idea is based on it.

The algorithm starts by finding all points that are lower than points on either side of them in the time series (local troughs) and putting their time indices in a set \( V_1 \). All the points in the set \( V_1 \) are troughs. All troughs in the set \( V_1 \) are then checked, and those troughs that are lower than both observations two observations away either side of it in the time series, are put into set \( V_2 \). The same is repeated for set \( V_2 \): all troughs in \( V_2 \) that are lower than points three observations away in the time series, on either side of it, are put into set \( V_3 \). This continues until the stopping criterion is met, which will be discussed shortly.
2.4. BSA-EIISR

Since the troughs in the set $V_i$ are the only points that will be investigated after iteration $i$, it is clear that we have a decreasing sequence of sets

\[ \forall i \in \mathbb{N} : V_i \supseteq V_{i+1} \]  

(2.11)

$V_i$ will be referred to as the **clearance set** after iteration $i$, and the sequence in equation (2.11) as the sequence of clearance sets. Note that the clearance set $V_i$ contains all those troughs that are lowest in a window of radius $i$. These may also be referred to as troughs of order $i$. An illustration of the contents of each clearance set $V_i$ is shown in figure 2.4. Note that all troughs $x_j$ in $V_i$ satisfy

\[ y(x_j) \leq y(x_{j \pm k}), \text{ where } k = 1, 2, ..., i \]

The main purpose of sequentially finding troughs that are lowest in wider and wider windows is to make the set from which troughs are chosen for the final solution, as small as possible. This sequential process makes it easy to determine which window size yields a set of troughs that is approximately equal in size to the required number of troughs, $\frac{K}{2}$. After the set of possible troughs is made small enough, if the number of troughs in the final set is not $\frac{K}{2}$, the second approach is applied, which will then be feasible because of the smaller set of troughs. This entails evaluating all solutions corresponding to all combinations of $\frac{K}{2}$ troughs extracted out of the two final sets of troughs (with the highest point between each, selected as sell points) to find the best solution. This is explained in more detail below. If the final set contains $\frac{K}{2}$ troughs, no further selection is needed, thus, the final set then consists of the optimal troughs.

### 2.4.3 Stopping conditions

Assume that the first part of the algorithm (dealing with the first sub-problem) stops after $F$ iterations, in which case the set of troughs at the end of the first sub-problem is denoted by $V_F$. Keeping in mind that the sequence of clearance sets is a decreasing sequence of sets, if too few iterations are run, then the final clearance set will have too many troughs (i.e. many more than $\frac{K}{2}$) to choose from in the final clearance set $V_F$. If the number of troughs in the set $V_F$ is too large, then evaluating all possible combinations of $\frac{K}{2}$ troughs that define
2.4. BSA-EIISR

(a) Contents of $V_1$: All local troughs are in $V_1$, since local troughs are lower than 1 time series observations either side

(b) Contents of $V_2$: Only local troughs that are lower than at least 2 time series observations either side are included in $V_2$

(c) Contents of $V_3$: Only troughs in $V_2$ that are lower than at least 3 time series observations either side are included in $V_3$

(d) Contents of $V_4$

Figure 2.4: Each image shows the contents of a clearance set, represented by red dots
each solution could be computationally very expensive. As an example, consider the case
where \( V_F \) has 50 elements (i.e. \(|V_F| = 50\)) and \( K = 20 \). Since every combination of \( \frac{K}{2} = 10 \)
troughs must be tested as buy points,

\[
\binom{|V_F|}{\frac{K}{2}} = \frac{|V_F|!}{(|V_F| - \frac{K}{2})! \frac{K}{2}!} = \frac{50!}{40!10!} \approx 1.027 \times 10^{10}
\]

different solutions must be evaluated to determine which combination of troughs results
in the highest profit fitness. From this it is easy to see that the algorithm could become
computationally very expensive, especially if \(|V_F|\) is much larger than \( \frac{K}{2} \), and \( \frac{K}{2} \) is suffi-
ciently larger than 1. However, if an unnecessarily large number of iterations is specified,
computational resources will be wasted. An appropriate stopping condition must therefore
be specified.

The solution to the problem of when to stop the first part of the optimisation procedure
is relatively simple. If \(|V_i| \leq \frac{K}{2}\), all the troughs in the clearance set at iteration \( i \) and after
will be unconditionally included when selecting the \( \frac{K}{2} \) most prominent troughs. Also, the
troughs in \( V_i \) are the only points that the algorithm will check at least once in subsequent
iterations (after iteration \( i \)). It therefore follows that if \(|V_i| \leq \frac{K}{2}\), then no more information
on which troughs to extract could be obtained by continuing the optimisation process, since
there is no need to distinguish between troughs that will definitely be extracted anyway. The
first part of the optimisation procedure is therefore terminated when \(|V_i| \leq \frac{K}{2}\). If \(|V_F| = \frac{K}{2}\),
all the troughs in \( V_F \) are taken as the troughs in the final solution. Otherwise, if \(|V_F| < \frac{K}{2}\),
then, since the algorithm was stopped as soon as \(|V_i| \leq \frac{K}{2}\), it follows that \(|V_{F-1}| > \frac{K}{2}\).
Therefore we will also choose from the troughs in \( V_{F-1} \), since \( V_{F-1} \) is then the smallest set
that contains at least all the troughs that will be extracted (because \( V_F \) does not contain
enough troughs). However, all the troughs contained in \( V_F \) must be kept because the troughs
with the highest prominence are contained in this set; therefore \( N_{extract} = \frac{K}{2} - |V_F| \) troughs
must be selected from a subset of the set \( V_{F-1} \) and added to the troughs in \( V_F \) for a full
solution of \( \frac{K}{2} \) troughs. Since all the troughs in \( V_F \) are in the final solution, and \( V_F \subset V_{F-1} \),
the subset of \( V_{F-1} \) that \( N_{extract} \) troughs is extracted from, is defined as

\[
V_{extract} = V_{F-1} \setminus V_F
\]

Each combination of size \( N_{extract} \) extracted from \( V_{extract} \) is combined with all the troughs
in $V_F$ to form a full solution for each combination. There will therefore be $\binom{|V_{extract}|}{N_{extract}}$ different solutions that must be evaluated to obtain a final solution (the solution with the highest fitness out of the $\binom{|V_{extract}|}{N_{extract}}$ solutions generated from the troughs in $V_F$ and the combinations out of $V_{extract}$).

If the final iteration occurs after sufficient iterations $F$, we can generally assume that $V_{extract}$ will not be a large set (in appendix B on page 149, there is a discussion on why this is generally so). Therefore the different $\binom{|V_{extract}|}{N_{extract}}$ combinations of points to be tested will not be very large and can all be tested without a great sacrifice in computation time. Note that, as stated already, this assumes that the number of iterations $F$ is sufficiently large. If $F$ is relatively small (such as when $K$ is relatively large compared to the series size) then BSA-EIISR may take very long to finish computation due to the large number of combinations that must be evaluated. This is the main shortcoming of the algorithm.

### 2.4.4 Pseudocode

BSA-EIISR is described formally in algorithms 4 and 5 on pages 39 and 40 respectively. Note that in algorithm 5, the AllCombinations($A$, $c$) function returns the set of all combinations of size $c$ of elements in set $A$. For example, AllCombinations($\{1, 4, 5, 9\}$, 3) returns a set $P_c = \{\{1, 4, 5\}, \{1, 4, 9\}, \{1, 5, 9\}, \{4, 5, 9\}\}$. Each element of $P_c$ is a unique combination of size 3 of the set $\{1, 4, 5, 9\}$. The FindPeaks($V$) function finds the highest peak between all pairs of troughs described by the indices in set $V$ and between the final trough in $V$ and the end of the time series, as illustrated in figure 2.3, and returns an alternating sequence of those peaks and troughs, which comprises a full solution, $X^*$.
Algorithm 4 BSA-EIISR main procedure

1: procedure BSA-EIISR($K$, $Y$)
2:   Initialise $V_0 \leftarrow \{x_1, x_2, x_3, ..., x_{Y_{\text{size}}\}}$ 
3:      $\triangleright$ Clearance set initially contains all time series observations
4:   $i \leftarrow 0$
5:   while $|V_i| > \frac{K}{2}$ do
6:     $V_{i+1} \leftarrow \emptyset$
7:     for each $x_v \in V_i$ do
8:         if $y(x_v) < y(x_{v-(i+1)})$ and $y(x_v) < y(x_{v+(i+1)})$ then
9:            $V_{i+1} \leftarrow V_{i+1} \cup \{x_v\}$
10:       end if
11:   end for
12:   $i \leftarrow i + 1$
13: end while
14: $V_{\text{optimal}} \leftarrow \text{EXTRACTTROUGHS}(V_i, V_{i-1}, K)$
15: return $\text{FindPeaks}(V_{\text{optimal}})$
16: end procedure
Algorithm 5 BSA-EIISR procedure for extracting optimal troughs

1: \textbf{procedure} \textsc{ExtractTroughs}(V_{final}, V_{final-1}, K)
2: \hspace{1em} if $|V_{final}| < \frac{K}{2}$ then
3: \hspace{2em} $V_{extract} \leftarrow V_{final-1} \setminus V_{final}$
4: \hspace{2em} $N_{extract} \leftarrow \frac{K}{2} - |V_{final}|$
5: \hspace{2em} $\mathcal{P}_{N_{extract}} \leftarrow \text{AllCombinations}(V_{extract}, N_{extract})$
6: \hspace{2em} for each $V_{comb} \in \mathcal{P}_{N_{extract}}$ do
7: \hspace{3em} $V_{cand} \leftarrow V_{final} \cup V_{comb}$
8: \hspace{3em} $X \leftarrow \text{FindPeaks}(V_{cand})$
9: \hspace{3em} if $f_{\text{profit}}(X) > f_{\text{profit}}(X_{best})$ then
10: \hspace{4em} $X_{best} \leftarrow X$
11: \hspace{4em} $V_{best} \leftarrow V_{cand}$
12: \hspace{3em} end if
13: \hspace{2em} end for
14: \hspace{1em} return $V_{best}$
15: \hspace{1em} else
16: \hspace{2em} return $V_{final}$
17: \hspace{1em} end if
18: \textbf{end procedure}
2.5 BSA-BU

The following definition will simplify the explanation of the BSA-BU method, which we present here.

**Definition 2.3. Validity of buy-sell points**

*A sequence of buy-sell pairs* \((x_{2i}, x_{2i+1})\) *for* \(i = 1, \ldots, N\), *where* \(N\) *is an arbitrary integer,* that satisfies the constraints of the optimisation problem given by equation (2.1), *is a valid sequence of buy-sell pairs.* The same applies to pairs \((x_{2i-1}, x_{2i})\). *If a sequence of buy-sell pairs is not valid,* then it is said to be invalid.

Note that, by implication, a valid sequence of buy-sell pairs never overlap. That is, there are no pairs \((x_i, x_j)\) such that another pair \((x_k, x_i)\) is defined such that \(x_i \leq x_k \leq x_j\), or \(x_i \leq x_l \leq x_j\), or \(x_k \leq x_i \leq x_j \leq x_l\).

The representation of the profit fitness function in equation (2.1) presents it as successive buy-sell (or *long*) pairs \((x_{2i}, x_{2i-1})\). Given that \(y(x_{2i}) > y(x_{2i-1})\), \(\forall i = 1, 2, \ldots, \frac{K}{2}\) (i.e. each sell point is higher than the preceding buy point), it is likely that the \(\frac{K}{2}\) pairs with the largest vertical distance between them will give us an optimal or near-optimal solution. If there are \(P_S\) possible long pairs in the time series, then it follows that if the \(P_S - \frac{K}{2}\) smallest long pairs are eliminated, so that the remaining pairs are valid, then the remaining pairs are the \(\frac{K}{2}\) largest long pairs.

The profit fitness function can also be written as follows

\[
\hat{f}_{\text{profit}} = -y(x_1^*) + \sum_{i=1}^{\frac{K}{2} - 1} (y(x_{2i}^*) - y(x_{2i+1}^*)) + y(x_K^*) \tag{2.12}
\]

Assume that \(x_1^*\) and \(x_K^*\) are the first and last buy-sell points, and that these are already optimal. The sell-buy (or *short*) pairs \((x_{2i}^*, x_{2i+1}^*)\), with the largest vertical distance between them, given that \(y(x_{2i}^*) > y(x_{2i+1}^*)\), \(\forall i = 1, 2, \ldots, \frac{K}{2} - 1\) must now be found. We need to find \(\frac{K}{2} - 1\) such pairs (since the first two buy-sell points, \(x_1^*\) and \(x_K^*\), are already optimal), so if there are a total of \(P_T\) short pairs, \(P_T - \frac{K}{2} + 1\) short pairs with the smallest vertical distance between them must be eliminated to find the pairs that will approximate an optimal solution. As in the case above, the remaining pairs have to be valid.
The BSA-BU method is based on the reasoning shown above. The method starts by assuming that every peak (trough) in the time series is a sell (buy) point. Since it was shown above that by eliminating the smallest long pairs, as well as the smallest short pairs, an approximation to the optimal solution may be made, the method eliminates short pairs as well as long pairs, based on the vertical distance between them. Pairs with the smallest vertical distance between them are eliminated first, until a total of \( K \) buy-sell points remain. A simple example of how this works is shown in figure 2.5. Note that the name of the algorithm comes from the fact that this method’s approach is similar to the bottom-up approach for time series representation, described in Keogh et al. (2004, p. 9).

This manner of eliminating buy-sell points ensures that the sequence of buy-sell pairs is always valid. This can easily be shown with the representations in equations (2.1) and (2.12). If a pair with a buy point first, and a sell point second is eliminated, a pair from the summation in (2.1) is eliminated without affecting any of the buy-sell pairs in the summation. This is illustrated in figure 2.5c. If a pair with a sell point first and a buy point second is eliminated, then one of the pairs in the summation in (2.12) is eliminated, once again without affecting any of the other sell-buy pairs in the summation. This is shown graphically in figures 2.5a and 2.5b. It is therefore guaranteed that eliminating any pair from a valid sequence of buy-sell pairs will not make the sequence invalid. Since the BSA-BU method starts with a valid sequence of buy-sell or sell-buy pairs, it is guaranteed that the sequence \( X^* \) is valid at any point in time (i.e. any iteration) during the BSA-BU algorithm.

Note that since BSA-BU checks only a subset of all possible pairs, the solution found by BSA-BU is in most cases still an approximation of the true optimum. However, BSA-BU does not check any long pairs where a buy point is lower than a sell point or short pairs where a buy point is higher than a sell point (pairs that will not improve the objective function). Therefore, many of the pairs that BSA-BU does not check are pairs that would have been a waste of time to consider anyway. In this way, clearly inferior solutions are automatically ignored by BSA-BU, so that it saves a lot of time in obtaining a solution. Unfortunately, potentially more optimal solutions are also excluded in this process, along with those mentioned above.

Since BSA-BU starts with the highest number of buy-sell points possible in the series, and eliminates these until the desired number of points is achieved, it seems likely that BSA-BU will perform better when \( K \) is high. This is because less buy-sell points are eliminated with higher \( K \), which implies that less computation is needed (i.e. lower computation time).
Further, since BSA-BU starts by assuming all peaks and troughs in the time series are buy-sell points, this solution is exactly the optimal solution when $K$ is as high as possible (i.e. when $K$ is equal to the number of peaks and troughs in the series). When $K$ is lower, more buy-sell points are eliminated from BSA-BU’s solution, therefore it becomes more likely that the solution given by BSA-BU is different from the optimal solution as $K$ gets smaller, as there are then increased chances of the method eliminating the wrong buy-sell points. It is therefore likely that higher $K$ also benefits BSA-BU in terms of profit fitness (relative to other methods).

### 2.5.1 Stopping conditions

Assume there are $N^*$ buy-sell points in $X^*$ initially, where $N^*$ is the number of local peaks and troughs in the time series. Buy-sell points are eliminated until $K$ buy-sell points remain in $X^*$. In each iteration, 2 points get eliminated. Therefore BSA-BU runs for $\frac{N^* - K}{2}$ iterations.

### 2.5.2 Pseudocode

While the reasoning given above is relatively complex, the method itself is conceptually quite simple. The fundamental workings of the method are given more formally in algorithm 6.

#### Algorithm 6 BSA-BU algorithm

```
1: procedure BSA-BU($K$, $Y$)
2:     $\hat{Y} \leftarrow \text{PreprocessData}(Y)$
3:     $X^* \leftarrow \{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_{N^*}\}$ \quad $\triangleright$ $X^*$ consists of all local peaks and troughs in the series
4:     while $|X^*| > K$ do
5:         $\{\hat{x}_i, \hat{x}_{i+1}\} \leftarrow \text{WeakestPair}(X^*, \hat{Y})$ \quad $\triangleright$ returns the pair of successive buy-sell points in $X^*$ that has the lowest vertical distance
6:         $X^* \leftarrow X^* \setminus \{\hat{x}_i, \hat{x}_{i+1}\}$
7:     end while
8: end procedure
```
2.5. BSA-BU

(a) Iteration one: A sell-buy pair is eliminated

(b) Iteration two

(c) Iteration three: Elimination of a buy-sell pair

(d) Final solution

Figure 2.5: BSA-BU working example. The pair connected by the thick red line is the "weakest" pair to be eliminated in that iteration

2.5.3 Working example

A working example of BSA-BU is shown in figure 2.5. Four iterations are shown, with ten initial buy-sell points, indicated with red dots.
2.6 Particle swarm optimisation (PSO)

Particle swarm optimisation (PSO) is a stochastic population-based search method designed for solving difficult optimisation problems that are hard or impossible to solve by traditional methods such as gradient descent. More particularly, PSO falls under the category of optimisation methods known as swarm intelligence (SI). SI is based on the collective intelligence of large groups of agents that communicate with each other by acting on their local environment (Engelbrecht 2007, p. 285).

The basic idea behind PSO is that there are multiple agents (known as particles) that search the environment, or domain of the objective function, for an optimal solution. The domain here is given as a $K$-dimensional space through which these particles move in search of a best solution. The “best solution”, is determined by the optimisation problem stated in equation (2.1), therefore $f_{profit}(X^*) > f_{profit}(X^{**})$ indicates that $X^*$ is a superior solution to $X^{**}$.

The basic components of the PSO algorithm are as follows

- Fitness, or objective function;
- Population (or swarm) of particles;
- Each particle $i$ has
  - An $n$-dimensional position (or candidate solution) $X^*_i$. Here $n$ is the number of variables in the objective function (therefore $n = K$ in our case);
  - An $n$-dimensional velocity $v_i$ (also $n = K$ in our case). The velocity gets added to the position at iteration $t$ to determine the position at iteration $t + 1$;
  - A personal best at iteration $t$
    * the best position this particle has visited until iteration $t$;
    * influences the velocity at iteration $t$;
  - its own neighbourhood of other particles;
  - a neighbourhood best at iteration $t$. This is the best position that any particle in the neighbourhood has visited so far (up until iteration $t$);
- cognitive scaling factor: a parameter that controls magnitude of the influence that the personal best has on velocity;
• social scaling factor: a parameter that controls magnitude of the influence that the neighbourhood best has on velocity.

Note that, regarding the concept of neighbourhoods discussed above, it is possible for each particle’s neighbourhood to be defined as the whole swarm (of particles). This specialisation of PSO is known as \textit{global best PSO}. When each particle’s neighbourhood is a subset of the swarm, the method is known as \textit{local best PSO}. This is discussed in some more detail in section 2.6.2.

\begin{algorithm}
\caption{Generic PSO algorithm}
\begin{algorithmic}[1]
\Procedure{GenericPSO}{$n$, $m$, \textit{StoppingCondition}, $f(.)$}
\Repeat
\For{each particle $i \leftarrow 1, 2, ..., m$}
\If{$f(X_i^*) > f(\bar{X}_i^*)$}
\State $\bar{X}_i^* \leftarrow X_i^*$ $\triangleright$ set personal best
\EndIf
\If{$f(\bar{X}_i^*) > f(Z_i)$}
\State $Z_i \leftarrow \bar{X}_i^*$ $\triangleright$ set neighbourhood best
\EndIf
\EndFor
\For{each particle $i = 1, 2, ..., m$}
\For{each dimension $j = 1, 2, ..., K$}
\State $v_{ij} \leftarrow$ updated velocity for dimension $j$
\State $x_{ij}^* \leftarrow x^*_{ij} + v_{ij}$ $\triangleright$ update position $X_i^*$
\EndFor
\EndFor
\Until{\textit{StoppingCondition} \texttt{true}}
\EndProcedure
\end{algorithmic}
\end{algorithm}

Adapted from Engelbrecht (2007, pp. 291, 293)

The velocity update for particle $i$ in dimension $j$ is given by the following equation

$$v_{ij} \leftarrow \alpha v_{ij} + r_{1j} s(z_{ij} - x_{ij}^*) + r_{2j} c(\bar{x}_{ij}^* - x_{ij}^*)$$ \hspace{1cm} (2.13)

where
2.6. PARTICLE SWARM OPTIMISATION (PSO)

- $1 \leq j \leq K$;
- $v_{ij}$ is the velocity in dimension $j$ for particle $i$;
- $\alpha$ is the inertia weight;
- $r_{1j}, r_{2j} \sim U(0, 1)$ and are independent;
- $s$ and $c$ are acceleration coefficients; $s$ will be referred to as the social scaling factor and $c$ as the cognitive scaling factor;
- $z_{ij}$ is dimension $j$ of the global best position (in case of global best PSO) or the best position in particle $i$’s neighbourhood (in case of local best PSO);
- $\bar{x}_{ij}^*$ is dimension $j$ of the current best position that particle $i$ has obtained so far.

To update the position for particle $i$, particle $i$’s velocity is added to its current position in each dimension $j$, as shown below

$$x_{ij}^* \leftarrow x_{ij}^* + v_{ij} \quad (2.14)$$

Since this method is applied here to a discrete domain (since $f_{profit}$ is only defined on discrete spaces), and the mechanisms above inherently work with continuous spaces, each particle’s position will be rounded to the closest integer after it was updated with equation (2.14). Generally, in PSO, many iterations are run (at least a few hundred in our case) and this may eventually have a large effect on the result obtained by the algorithm, since rounding is applied at each iteration. Therefore, the velocity will not be rounded in an effort to keep the effects of rounding as low as possible. It would have been possible to round the position only temporarily when evaluating it with $f_{profit}$ and letting it keep the unrounded values afterward, but that could have caused some complications regarding the handling of the constraints. Only the positions were therefore rounded, without reverting to unrounded values afterward.

Note that each particle’s position in dimension $i$ corresponds to the time series index of buy-sell point $i$. The constraint $x_i^* < x_j^*$ if $i < j$, shown in section 2.1, means that, if we store buy-sell points in a list, where the buy-sell point index $x_i$ is at index $i$ of the list, then this list must remain sorted from smallest to largest. However, in PSO this constraint is very likely to be violated (the previous methods presented inherently kept this ordering to
the buy-sell points). Below, various options to deal with this restriction are discussed, after which the parameters of the algorithm are discussed.

2.6.1 Constraint handling

Some options to deal with the constraints are as follows

- To leave the core PSO algorithm unaltered, and modify only the fitness function for each particle such that it gets heavily penalised if it fails to meet at least one of the constraints. This is also known as adding a penalty term to the objective function. In particular, a particle that falls outside the feasible area (even if it violates only one of the constraints) can have its fitness value evaluated to $-\infty$ to prevent any position outside the feasible area to become a local or global best position. Since it is easy to generate the initial positions for all particles such that they all fall within the feasible region, global and local bests will always be in the feasible region. This implies that particles are always drawn to the feasible region, even if they may be in an infeasible region temporarily. However, this approach may waste computational resources and make the algorithm less efficient, since then particles spend time in regions where no feasible solutions exist.

- To simply order the list of buy-sell points from smallest to largest. However, this approach might have the undesired effect of particles moving in opposite directions from their intended direction in some dimensions (particularly when two buy-sell points are “swapped” in the ordering). This can also make the algorithm slower, since sorting can be an expensive operation (computationally), especially if a large number of unordered elements are to be sorted.

- A. Kaur and M. Kaur (2015, p. 15) suggest utilising velocity clamping to keep the particles inside the feasible region by keeping their velocities within ranges that would prevent them from entering into infeasible regions. However, the problem with this approach is that, since the range of possible values for at least one dimension changes as soon as it changes position in a single dimension (because of the constraint $x_i < x_j$ if $i < j$), velocity clamping must be applied each time an update is made in a dimension. On a practical level, this would have prevented the use of efficient vectorised operations in Python (using Numpy’s vector addition operations), since a custom function would
have had to be written to apply this type of clamping and position update at the same time. An alternative is to apply velocity clamping at once, without updating positions each time a velocity is clamped. Positions are then afterwards updated all at once (using Numpy’s fast built-in vector addition operations). In this case, velocity clamping does not prevent a particle from entering the infeasible region; however, since velocities are still kept under control in this case, particles would likely not be far inside the infeasible regions.

A combination of the second and third approaches were used. Before each position was updated, velocity clamping was applied to all dimensions at once. After this, all dimensions (or buy-sell points) in the position were updated at once. Due to the nature of the constraints, it is possible for particles to be in infeasible regions after this step; sorting was therefore applied, so that the particle may be kept inside the feasible region. Since the particles are more likely to be not far from the feasible region, it is expected that the amount of sorting to be done is also reduced.

Note that the sorting function used is a built-in Python function. Built-in functions for programming languages are generally faster, since they are usually optimised. For this reason, it was considered preferable to use a built-in sorting function, than to use custom functions to update positions and apply velocity clamping at the same time.

**Velocity clamping: details**

For dimensions 2 to \( K - 1 \) (where \( K \) is the total number of dimensions) the following function was used for clamping velocities

\[
v_{ij} = \begin{cases} 
  x^*_{i,j+1} - x^*_{i,j} - 1 & \text{if } x^*_{i,j} + v_{ij} > x^*_{i,j+1} - 1 \\
  x^*_{i,j-1} - x^*_{i,j} + 1 & \text{if } x^*_{i,j} + v_{ij} < x^*_{i,j-1} + 1 \\
  v_{ij} & \text{otherwise}
\end{cases}
\]

Therefore \( v_{ij} \in [x^*_{i,j-1} - x^*_{i,j} + 1, x^*_{i,j+1} - x^*_{i,j} - 1] \) for \( j = 2, 3, ..., n - 1 \).

For dimension 1 (i.e. point \( x^*_{i,1} \)), the adjustment was as follows
2.6. PARTICLE SWARM OPTIMISATION (PSO)

\[ v_{i1} = \begin{cases} 
 x_{i,2}^* - x_{i,1}^* - 1, & \text{if } x_{i,1}^* + v_{i1} > x_{i,2}^* - 1 \\
 1 - x_{i,1}^*, & \text{if } x_{i,1}^* + v_{i1} < 1 \\
 v_{i1}, & \text{otherwise}
\end{cases} \]

Therefore \( v_{i1} \in [1 - x_{ij}^*, x_{i,j+1}^* - x_{ij}^* - 1] \).

Finally, for dimension \( K \) (i.e. point \( x_{i,K}^* \))

\[ v_{iK} = \begin{cases} 
 N^* - x_{i,K}^* - 1, & \text{if } x_{i,K}^* + v_{iK} > N^* \\
 x_{i,K-1}^* - x_{i,K}^* + 1, & \text{if } x_{i,K}^* + v_{iK} < x_{i,K-1}^* + 1 \\
 v_{iK}, & \text{otherwise}
\end{cases} \]

Therefore \( v_{iK} \in [x_{i,K-1}^* - x_{i,K}^* + 1, N^* - x_{i,K}^*] \), where \( N^* \) is the length of the reduced representation of the time series (only the peaks and troughs).

2.6.2 Parameters

All parameters discussed below, except the cognitive and social scale parameters, will be optimised for each type of problem (defined by the series size, Hurst exponent, and number of buy-sell points, \( K \)) that PSO is applied to. For that reason, specific values for only the cognitive and social scale parameters are given below, while other parameters are discussed in more general terms.

Swarm size

This refers to the amount of particles used in the optimisation procedure. Generally, the larger the swarm, the fewer iterations are needed to obtain a solution of the same quality as obtained when having a smaller number of particles. However, since more particles must be processed in an iteration, each iteration will take longer to finish if swarm size is larger. On the other hand, a larger number of particles generally reduces the risk of the PSO algorithm getting “stuck” in a local optimum, since then there is usually more diversity in the swarm.

Chen, Montgomery, and Bolufé-Röhler (2015, p. 525) recommends that the swarm size
be kept lower than the number of dimensions ($K$ in our case) if number of dimensions are high. Chen, Montgomery, and Bolufé-Röhler (2015, p. 515) classify a “high” number of dimensions as more than 50 (i.e. $K > 50$). The only value of $K$ investigated that is higher than 50 is $K = 100$; therefore swarm size was limited to 99 particles in all cases where PSO was applied in this study, even for lower values of $K$. More details on how swarm sizes were determined are given in chapter 4.

**Cognitive scaling factor**

The cognitive scale parameter controls the effect of the particle’s own domain knowledge on its behaviour. A higher value (keeping the social scale parameter, discussed below, constant) favours exploration, since then each particle will most likely follow a unique location and more solutions will be explored. Lower values imply that the algorithm becomes more greedy, which means that existing solutions are more likely to be refined, rather than new solutions explored.

Ratnaweera, H algamuge, and Watson (2004, p. 246) recommend that this parameter be set at 2.5 at the start of the optimisation procedure, and decreased linearly such that it is at 0.5 at the end of the procedure. This allows particles to explore more regions of the optimisation landscape at the start of the algorithm, then gradually starts refining solutions already found. This parameter was varied in the above-mentioned manner.

**Social scaling factor**

This parameter controls the effect of the particle’s neighbours’ domain knowledge on its behaviour. A higher value (keeping the cognitive scale parameter constant) implies exploitation (greediness), since then each particle will move towards a location that a number of other particles are also moving to (this “number of other particles” is defined by the neighbourhood size, discussed below), while lower values imply that the algorithm will become less greedy.

In much the same way as with the cognitive scale parameter above, Ratnaweera, H algamuge, and Watson (2004, p. 246) recommend that this parameter be set at 0.5 at the start of the optimisation procedure, and increased linearly so that it is at 2.5 at the end of the procedure. The reasoning behind this is the same as in the cognitive scale parameter,
therefore this parameter was also varied in the same manner.

Neighbourhood size

Each particle is drawn to a certain position with the extent determined by the social scale parameter. This position that each particle is drawn to is defined as the best position in the neighbourhood of each particle. In the global best PSO, each particle’s neighbourhood is the whole swarm. In each iteration then, each particle will be drawn to the best position visited thus far by any particle in the whole swarm. In the local best PSO, each particle’s neighbourhood is a subset of the swarm, and it is drawn to the best position visited thus far by any particle in this subset. There are a few ways in which a neighbourhood can be defined. Some common examples are shown in figure 2.6, where a particle is represented by a dot, and a particle’s neighbourhood is all the particles that are connected directly to it by the black lines. Note that in this study “neighbourhood size” is defined as the amount of particles that each particle has in its neighbourhood (excluding itself).

The star and ring topologies were used for all implementations of PSO in this study. In the case of the ring topology, the neighbourhood size can be varied from two (shown in figure 2.6c) to larger sizes (as in figure 2.6d, with the largest possible neighbourhood shown in figure 2.6e). The benefit of using the ring topology in this way is that there is a spectrum on which one may specify how quickly information on good solutions spreads through the swarm (i.e. how quickly convergence takes place on average). When neighbourhood sizes are small, convergence generally takes longer, which favours exploration. On the other hand, when neighbourhood sizes are large, exploitation is favoured, which implies that convergence to a solution is generally faster. Neighbourhoods for each particle remains constant throughout the execution of each PSO algorithm. That is, each particle shares information directly with the same set of particles regardless of the number of iterations that have passed, or position of each particle.

We understand that this ability to position the speed of convergence along this spectrum will benefit us, since there is reason to believe that some of the time series that the optimisation procedure is applied on will have more local maxima than others, requiring a swarm that will favour exploration over exploitation. This is illustrated by the fitness landscapes shown in figure 2.7. Note that the illustrations in figure 2.7 are only for 2-dimensional optimisation (i.e. when $K = 2$). More often, when $K$ is higher, the fitness landscapes will be much more
simple and multi-modal than those shown in figure 2.7.

The neighbourhood size will be specified as a ratio of the swarm size. More specifically, if neighbourhood size is given by \( n \) and swarm size by \( P_{PSO} \), then the neighbourhood ratio is given by the following

\[
    n_{\text{ratio}} = \frac{n}{P_{PSO}}
\]

The size of the neighbourhood is specified in this way because, if a swarm size of ten with a neighbourhood size of four is used, then each particle shares information with almost half of the swarm (i.e. neighbourhood ratio is 0.4), which implies that convergence will be fast. If swarm size is 90, also with a neighbourhood size of four (which implies a neighbourhood ratio of about 0.044), then the spread of information, and thus convergence, is slower than in the first case. Neighbourhood ratio is therefore a more objective measure of how much the neighbourhood size affects the speed of convergence.

**Inertia weight**

This parameter keeps velocity under control, especially in cases where a particle is oscillating over a search region. If this coefficient is equal to one or more, particles may oscillate wildly when constantly attracted by a certain search region. When it is less than one, particles attracted to the same solution for a period of time will slow down, enabling the algorithm to refine its current solutions.

Bansal et al. (2011, pp. 635 - 638) investigated various ways in which the inertia weight in PSO may be determined or adapted. They reported that when applied to minimise the Rastrigin function, PSO with constant inertia weight 0.7 obtained on average the minimum difference between the solution found by PSO and the true optimum, when compared to other methods for adapting the inertia weight (i.e. other methods that vary the inertia weight, so that it is non-constant during optimisation). The Rastrigin function, given by

\[
f(X) = 10n + \sum_{i=1}^{n} [x_i^2 - 10cos(2\pi x_i)]
\]

has many local optima, resembling many of the cases PSO is applied to here. However, an
2.6. PARTICLE SWARM OPTIMISATION (PSO)

(a) Wheel topology: All information on local best positions is communicated through a single particle.

(b) Von Neumann: Particles are connected in a lattice-based structure with 4 - 6 connections.

(c) Ring topology, neighbourhood size: 2. Each particle is connected to two adjacent particles. The slowest spreading of information throughout the swarm occurs in this case.

(d) Ring topology, neighbourhood size: 6. Each particle is connected to 3 particles on either side. Clearly, there is a greater degree of interconnectivity in the swarm, leading to faster spreading of information.

(e) Star topology (global best PSO). Conceptually, each particle is connected to every other particle, since the whole swarm acts as one large “neighbourhood”. Fastest spreading of information.

Figure 2.6: Different degrees of connectivity can be controlled with different swarm sizes.
2.6. PARTICLE SWARM OPTIMISATION (PSO)

(a) Typical fitness landscape with high Hurst exponent

(b) Typical fitness landscape when Hurst exponent is 0.5 (a Gaussian random walk). Notice there are four high local optima, with many jagged peaks throughout

(c) Side view of figure 2.7b

(d) Fitness landscape with low Hurst exponent. Also more than one local optima, but more jagged peaks over the whole landscape

Figure 2.7: Influence of Hurst exponent on the shape of the fitness landscape when $K = 2$. Note that the flat region in each graph indicates the infeasible region (i.e. where $x_1^* \geq x_2^*$)
inertia weight of 0.7 is used only as a guide for what inertia weights to use in this study, since Bansal et al. (2011, pp. 635 - 638) did not compare other constant values for inertia weights. The method by which inertia weights were determined will be discussed in chapter 4.

**Computation time**

The time (in seconds) that a PSO algorithm runs is predetermined by setting this parameter. Clearly, if the algorithm is given more time to finish, it is more likely to find better solutions. Allowing this parameter to vary will allow us to compare PSO with more computation time with PSO that had less computation time, so that the effect of allowing more or less computation time may also be assessed. Details on how the computation time was determined are given in chapter 4.

**2.6.3 Stopping conditions**

Stopping conditions are dictated by the computation time parameter, which was discussed above.

**2.6.4 Initialisation of swarm**

Each particle $i$ had its position in dimension $j$ initialised as follows

$$x_{ij}^* \leftarrow r_{ij} \frac{N^*}{K} + (j - 1) \frac{N^*}{K}$$

where $r_{ij} \sim U(0, 1)$. Each buy-sell point $x_{ij}^*$ was therefore sampled from $U((j-1)\frac{N^*}{K}, j\frac{N^*}{K})$. As a result, buy-sell points were on average more or less equally spaced in the time series. This is due to the fact that, most likely, more optimal sets of buy-sell points are found at more or less equal intervals on average. This also allows us to control initial velocity in such a way that the ordering of buy-sell points will be less likely to be disturbed at the start of the optimisation procedure, as shown below.

Velocities were initialised as follows
where \( s_{ij} \sim U(0,1) \). Velocities were therefore distributed according to
\[ U(-\frac{1}{4 \frac{N^*}{K}}, \frac{1}{4 \frac{N^*}{K}}), \]
meaning that in each dimension, the initial velocity was maximum a quarter of the average distance between neighbouring buy-sell points. Velocities were initialised at non-zero values in order to increase the exploration characteristics of the swarm, since the initial positions were generated rather strictly, which means that initial positions were generated non-uniformly over the search space. Engelbrecht (2007, p. 297) recommends that velocities, if not initialised at zero, should be initialised with relatively small values in order to reduce the occurrence of particles leaving the feasible region of the search space. This is the reasoning that was followed here.

### 2.6.5 Synchronous vs asynchronous updates

The updates on the positions, velocities and fitness of each particle in the swarm may be done in two ways: synchronously or asynchronously.

With synchronous updates, the same information regarding neighbourhood bests is made available to all particles before updates are made. This is achieved by first updating all velocities, then positions and finally all fitness values.

On the other hand, with asynchronous updates each particle’s position and fitness value is updated immediately after updating its velocity. Slightly different information is therefore available to each particle in the swarm each time its velocity and position are updated.

In Rada-Vilela, M. Zhang, and Seah (2011), a comparison is made between synchronous and asynchronous PSO on various optimisation problems. Rada-Vilela, M. Zhang, and Seah (2011, p. 27 - 28) report strong evidence that synchronous PSO outperforms asynchronous PSO on various problems, but also that synchronous PSO may sometimes converge faster than asynchronous PSO. This may lead to premature convergence, since the objective function in this case may have multiple local maxima (see figure 2.7). However, since PSO is tested with various neighbourhood sizes, the faster speed of convergence may be offset by decreasing neighbourhood sizes, which will allow an evaluation of PSO’s performance with difference rates of convergence in any case. For this reason, synchronous PSO was not a limitation in terms of preventing faster convergence rates. Synchronous updates were therefore
implemented.

2.6.6 Working example

A working example of the PSO algorithm applied to dating cycles is shown in figure 2.8 on page 59.

2.7 PSO-SGA

Since PSO has an element of randomness in the magnitudes by which it adjusts each buy-sell point, there is a reasonable probability that the algorithm will give us solutions where a sell (buy) point might be close to a peak (trough) that will improve the solution, but not quite on it. For this reason, after the PSO algorithm is run, PSO’s output (the final set of buy-sell points returned by PSO) may be used as input into SGA (the initial set of buy-sell points that SGA adjusts). In other words, instead of SGA generating a random initial set of buy-sell points, it will do its adjustments on the buy-sell points that PSO found, so that any buy-sell points found by PSO that narrowly missed a “better” peak or trough might be adjusted accordingly. Since the SGA algorithm is very fast, especially when its population size is low (shown in chapter 4), this final adjustment to PSO’s output incurs negligible computational cost (especially compared to how long PSO normally runs). This hybrid of methods will be referred to as PSO-SGA.

Note that results from both the original PSO algorithm and the PSO-SGA method are shown in the results section. This is done by extracting the solution from the PSO algorithm first, after which SGA uses that solution to obtain a second solution, which is the solution found by PSO-SGA.
Figure 2.8: Locations of a single particle at various iterations in the PSO algorithm. Note, for the first 300 iterations, the particle’s position (the set of buy-sell points, indicated by red dots) varies a lot, and does not seem to find or stay at any “good” solution. This is the exploration phase that the particle is going through, when it searches for as many different solutions as possible. At the 440th iteration, and after, the particle stays in one position and varies its position only slightly. This is the exploitation phase, where the particle refines a solution.
2.8 Summary

In this chapter, reasonably comprehensive descriptions of all the methods compared in this study were given. The reader should now be familiar with the rationale behind each algorithm, as well as the parameters of each, which form part of the comparison in chapter 4. In the next chapter, the methodology for comparing these algorithms, as well as details on the data that they were applied to, are given.

Note that in the following chapters, BSA-BU and BSA-EIISR will be referred to as BU and EIISR, respectively. These are not to be confused with the bottom-up algorithm given in Keogh et al. (2004, p. 2) and the EIISR method in Wu and Huang (2009, pp. 501 - 504). This shortened form is used purely for convenience; the two algorithms newly presented in this study are still formally known as BSA-BU and BSA-EIISR.
Chapter 3

Research methodology

In this chapter, we start by defining the optimisation problem in more detail. Information on the simulated time series data used, such as how it was generated, which stochastic processes were used to generate it, and how the parameters of these processes were decided, is then provided. Lastly, the statistical methods used to compare and determine the most efficient algorithms for the optimisation problem (among those presented in the previous chapter), as well as finding optimal parameters for these algorithms, are discussed.

3.1 Optimisation problem: objective function

3.1.1 Profit objective function

The profit objective function of the optimisation problem, equation (2.1), is restated below for ease of reference

\[ f_{\text{profit}}(X^*) = \sum_{i=1}^{n/2} y(x_{2i}^*) - y(x_{2i-1}^*) \]  

(3.1)

where \( y(x) \) is the asset price at time \( x \), and \( X^* \) is a set of the times of buy-sell points, i.e. \( X^* = \{x_1^*, x_2^*, \ldots, x_n^*\} \).

Note that as an alternative notation, the above objective function may also be represented
in terms of the specific execution of a particular algorithm, as shown below

$$g_{\text{profit}}(C_{ij}) = f_{\text{profit}}(X^*)$$  \hspace{1cm} (3.2)

where $C_{ij}$ refers to the j-th execution of algorithm $C_i$, and $f_{\text{profit}}(X^*)$ is defined as in equation (3.1). This notation will prove useful since a single objective function value is always obtained for a single execution of an algorithm. $X^*$ above refers to the solution (set of buy-sell points) that was obtained by $C_{ij}$.

The above profit fitness function only takes long positions into account. That is, where the sequence of alternating buy-sell points starts with a buy point, and ends with a sell point. To simplify the comparisons that were made, short positions (i.e. where the sequence starts with a sell point and ends with a buy point) were not considered. The algorithms could easily be modified to accommodate short selling, however, by multiplying the above objective function by negative one. Alternatively, equation (3.1) could also be minimised instead of maximised, subject to the same constraints, in order to accommodate short selling.

### 3.1.2 Time objective function

Since lower values of time (in seconds) is better than higher values of time taken to obtain a solution, the time objective function was defined as the inverse of the actual time (in seconds) to finish computation. This was done so that with both objective functions, higher values imply more desirable qualities. This made it easier when doing comparisons between different methods.

The time objective function is therefore as follows

$$g_{\text{time}}(C_{ij}) = \frac{1}{t(C_{ij})}$$  \hspace{1cm} (3.3)

where $C_{ij}$ is defined as in equation (3.2) above and $t(C_{ij})$ is time in seconds of the j-th execution of algorithm $C_i$.

Similar to the profit objective function, the time objective function may also be referred to as the “time fitness function” and the value of the time objective function as “time fitness”. However, if only the term “fitness” is used, it will generally refer to the profit fitness.
3.2 Time series simulation

In this section, the theory and methodology used to simulate time series data sets that the algorithms described in chapter 2 were applied to, is discussed. Note that the time series was generated to mimic stock price processes. We therefore drew from the literature on derivatives pricing, in particular the Black-Scholes model, which assumes some underlying random process for share prices. This underlying process was approximated in order to generate time series that resemble real-world, high sampling frequency share price data. Details are also provided on the parameters of the time series, which will be used to differentiate between time series when analysing results in chapter 4, after which discussions and more detailed properties of the stochastic process approximated are given.

3.2.1 Hurst exponent

The Hurst exponent is a measure that describes certain characteristics of any given time series. Specifically, it provides information on the long-term memory and fractality of a time series (Qian and Rasheed 2004, p. 203). It is generally denoted by $H$ and can assume any value between 0 and 1.

With $H$ values in this range, Qian and Rasheed (2004, p. 203) also state that a series can be broadly classified in three categories: for $H < 0.5$, we have an anti-persistent series, meaning an upward (downward) movement in the price/quantity that the series describes is more likely to be followed by a downward (upward) movement in the given quantity, which gives the series a more jagged appearance; for $H = 0.5$, the series is a random walk; for $H > 0.5$, the series is described as persistent, meaning an upward (downward) movement is more likely to be followed again by an upward (downward) movement, which gives the series a smoother appearance. The Hurst exponent is further illustrated by the graphs in figure 3.1.
3.2. TIME SERIES SIMULATION

Clearly, different values of $H$ seem to change the frequency and character of major peaks and troughs in the time series, which most likely has an effect on the nature of the buy-sell points extracted. Therefore, with regard to the performance of the algorithms on different time series, the Hurst exponent is one of the quantities that were investigated.
3.2.2 Continuous-time fractional Brownian motion

A short description of fractional Brownian motion (fBm) is given here. Fractional Brownian motion forms the basis of the process used to generate time series for this study.

The following definition of a Gaussian process was adapted from Revuz and Yor (1991, p. 34). Some notation has been changed, and additional observations have been left out from the original definition.

**Definition 3.1** (Gaussian process). A real-valued process, $B_t$, $t \in T$, is a Gaussian process if for any finite sub-family $(t_1, ..., t_n)$ of $T$, the random vector $(B_{t_1}, ..., B_{t_n})$ is Gaussian.

The following discussion was adapted from Nualart (2007, p. 1542). The symbols used here are the same used by the author.

**Definition 3.2** (Fractional Brownian motion). A Gaussian process $B^H = \{B^H_t; t > 0\}$ is called fractional Brownian motion, or fBm with Hurst parameter $H \in (0, 1)$ if it has mean zero and covariance function

$$E(B^H_t B^H_s) = \frac{1}{2}(s^{2H} + t^{2H} - |t - s|^{2H}) \quad (3.4)$$

Furthermore, it has the following properties:

**Self-similarity:** For any constant $a > 0$, the processes $\{a^{-H}B^H_{at}; t > 0\}$ and $\{B^H_t; t > 0\}$ have the same probability distribution. This property is an immediate consequence of the fact that the covariance function given by (3.4) is homogeneous of order $2H$, and it can be considered as a “fractal property” in probability.

**Stationary increments:** From (3.4), it follows that the increment of the process in the interval $[s, t]$ has a normal distribution with zero mean and variance

$$E((B^H_t - B^H_s)^2) = |t - s|^{2H} \quad (3.5)$$
3.2.3 Discrete-time fractional Brownian motion

In this study, all fBm processes were generated with sampling interval 1 (no particular unit of time is assumed), and is thus not continuous, but discrete. This does not affect the derivation of equation (3.4), despite its derivation assuming continuity, since one may still assume that the data is continuous, but that observations are sampled at 1-unit intervals. The only modification that we make is that we assume \( t, s \in \mathbb{N} \) in equations (3.4) and (3.5).

3.2.4 The fractional Black-Scholes model

Below, the exact process utilised for the simulation of all time series used in this study is derived. The rationale behind using this process is also given.

The following definition was adapted from Seydel (2012, p. 26). Some additional clarification in the original definition has been omitted.

**Definition 3.3** (Wiener process). A Wiener process (or standard Brownian motion; notation \( W_t \) or \( W \)) is a time-continuous process for \( t \geq 0 \) with the following properties

- \( W_0 = 0 \)
- \( W_t \sim N(0, t) \) for all \( t \geq 0 \).
- All increments \( \Delta W_t = W_{t + \Delta t} - W_t \) on non-overlapping time intervals are independent.
- \( W_t \) depends continuously on \( t \).

Many authors do not distinguish between Brownian motion and the Wiener process. Here, Brownian motion (as opposed to standard Brownian motion) is defined as follows.

**Definition 3.4** (Brownian motion). The stochastic process

\[
Z_t = Z_0 + \mu t + \sigma W_t
\]

is known as Brownian motion with drift \( \mu \) and volatility \( \sigma \).
Therefore, Brownian motion is a generalisation of standard Brownian motion, where \( Z_0, \mu \) and \( \sigma \) are allowed to vary, i.e. \( Z_0, \mu, \sigma \in \mathbb{R}, \sigma > 0 \).

The following definition is from Ross (2011, p. 38).

**Definition 3.5** (Geometric Brownian motion). Let \( Z_t \) be a Brownian motion process with drift parameter \( \mu \) and variance parameter \( \sigma^2 \), and let

\[
S_t = S_0 e^{Z_t}, \quad t \geq 0
\]

The process \( S_t, t \geq 0 \) is said to be a geometric Brownian motion process with drift parameter \( \mu \) and variance parameter \( \sigma^2 \).

In the modeling of asset prices, a geometric Brownian motion model of the following form is often assumed.

\[
S_t = S_0 e^{(\mu - \frac{\sigma^2}{2}) t + \sigma W_t}
\]  \hspace{1cm} (3.6)

In Bender, Sottinen, and Valkeila (2011, p. 77), the fractional Black-Scholes (fBS) model is investigated, which assumes the following process for asset prices

\[
S_t = S_0 e^{\mu t + \sigma B^H_t}
\]  \hspace{1cm} (3.7)

If we assume that an asset price process \( S_t \) follows the process given in equation (3.7), then taking the logarithm of this process yields the following

\[
\log(S_t) = Y_t = \log(S_0) + \mu t + \sigma B^H_t
\]  \hspace{1cm} (3.8)

which is an fBm process. This is the process simulated in this paper, which is assumed to be the natural logarithm of some asset price process. Using the logarithm of prices has certain benefits:

- Assuming a geometric model, the log price series is generated by a much simpler process than the geometric process. Furthermore, if the methods in chapter 2 were to be
applied to real data, then finding the logarithm of actual prices in order to make the data consistent with the time series model shown above in equation (3.8) is fairly trivial.

- In the log-price model, the drift and volatility parameters have a linear instead of exponential effect on the series. This gives changes in parameters a much simpler effect on the simulated data. Although drift and volatility will not be factors in this study (as shown below), this might make future studies easier where the effects of drift, together with volatility, are investigated.

- Yields between different prices become additive instead of multiplicative. This means that a difference of one unit between two log prices always implies the same yield, regardless of their levels. As shown in chapter 2, the optimisation methods applied here work by finding the largest additive rises and drops in share prices. Therefore if these methods are applied to log prices, then yields, instead of absolute differences between actual prices, are automatically taken into account. This is very desirable in the case of financial time series, since investors always focus on yields, rather than absolute differences in prices.

- Log prices, regardless of whether they are positive or negative, always imply positive actual asset prices. This made time series much easier to generate, since no restriction needed to be imposed in order to ensure that prices are positive, which would have been the case if actual prices were simulated.

For the process given by equation (3.8), we will assume $\mu = 0$. If this assumption is made, then changes in $\sigma$ have no effect on the performance of any algorithm used in this study. This is shown below.

**Theorem 3.1.** Assume a time series $Y$, such that $\mu = 0$, and a sequence of solutions $X_0^*, X_1^*, ..., X_n^*$ from dating cycles in time series $Y$, such that $f_1(X_0^*) \leq f_1(X_1^*) \leq ... \leq f_1(X_n^*)$, where $f_1(.) = f_{\text{profit}}(.)$. If the time series $Y$ were scaled to obtain a new time series $Y_t^* = \sigma_0 Y_t = \sigma_0 y(t) = \sigma_0 B_t^H$, then the sequence of solutions $X_0^*, X_1^*, ..., X_n^*$, with fitness values $f_2(X_0^*), f_2(X_1^*), ..., f_2(X_n^*)$ will still have the same order $f_2(X_0^*) \leq f_2(X_1^*) \leq ... \leq f_2(X_n^*)$ after dating the new scaled time series.
3.2. TIME SERIES SIMULATION

**Proof** Assume that a time series is given by equation (3.8)

\[ Y_t = \log(S_0) + \mu t + \sigma B_t^H \]

where \( \mu = 0 \).

Without loss of generality, we may assume that \( S_0 = 1 \Rightarrow \log(S_0) = 0 \), and also that \( \sigma = 1 \), therefore

\[ Y_t = B_t^H = y(t) \]

Any solution \( X^* \) has fitness function given by

\[ f_1(X^*) = \sum_{i=1}^{n/2} y(x^*_{2i}) - y(x^*_{2i-1}) \]

If the same time series \( Y_t \) was generated with \( \sigma_0 \) as volatility, i.e.

\[ Y_t^* = y(t)^* = \sigma_0 B_t^H = \sigma_0 y(t) \]

then the fitness function of the solution \( X^* \) has the following value

\[ f_2(X^*) = \sum_{i=1}^{n/2} \sigma_0 y(x^*_{2i}) - \sigma_0 y(x^*_{2i-1}) = \sigma_0 \sum_{i=1}^{n/2} y(x^*_{2i}) - y(x^*_{2i-1}) = \sigma_0 f_1(X^*) \]

Now, \( f_1(X_0^*) \leq f_1(X_1^*) \leq ... \leq f_1(X_n^*) \Rightarrow \sigma f_1(X_0^*) \leq \sigma f_1(X_1^*) \leq ... \leq \sigma f_1(X_n^*) \), from which it follows that

\[ f_2(X_0^*) \leq f_2(X_1^*) \leq ... \leq f_2(X_n^*) \]

\[ \square \]

Therefore any sequence of solutions will keep their relative fitness values after a time series is scaled.
Theorem 3.2. If \( y(t_0) > y(t_1) \) in time series \( Y \) (the same series described in theorem 1), then if \( Y_i^* = y^*(t) = \sigma_0 Y_i = \sigma_0 y(t) \), then \( y^*(t_0) > y^*(t_1) \).

**Proof** \( y(t_0) > y(t_1) \) ⇒ \( \sigma_0 y(t_0) > \sigma_0 y(t_1) \) ⇒ \( y^*(t_0) > y^*(t_1) \). \(\square\)

Note that there is no ordering to the subscript on \( t \), i.e. if \( i < j \) then it is not necessarily true that \( t_i < t_j \). This convention is maintained for the rest of the section. Note we still assume that if \( i < j \), then \( x_i < x_j \) (i.e. it is only for \( t_i \) that we relax this constraint).

Theorem 3.3. Let \( D(t_0, t_1) = |y(t_0) - y(t_1)| \), where \(|c|\) denotes here the absolute value of a real number \( c \). Now assume that we have a sequence of such absolute differences \( D(t_{00}, t_{01}), ..., D(t_{n0}, t_{n1}) \) such that \( D(t_{00}, t_{01}) \leq D(t_{10}, t_{11}) \leq ... \leq D(t_{n0}, t_{n1}) \). Now, if such differences were applied to the scaled time series \( Y_i^* \), which creates the sequence of absolute differences \( D(t_{00}, t_{01})^*, ..., D(t_{n0}, t_{n1})^* \) (where \( D(t_0, t_1)^* = |y(t_0)^* - y(t_1)^*| \)), then \( D(t_{00}, t_{01})^* \leq D(t_{10}, t_{11})^* \leq ... \leq D(t_{n0}, t_{n1})^* \).

**Proof** Let \( D(t_{ab}, t_{ac}) = |y(t_{ab}) - y(t_{ac})| \), where \( c = b + 1 \). Now, assume the same differences (at the same time indices) are applied to the scaled time series \( Y^* \), then \( D(t_{ab}, t_{ac})^* = |\sigma_0 y(t_{ab}) - \sigma_0 y(t_{ac})| \), from which it follows that \( D(t_{ab}, t_{ac})^* = \sigma_0 |y(t_{ab}) - y(t_{ac})| = \sigma_0 D(t_{ab}, t_{ac}) \). Therefore,

\[
D(t_{00}, t_{01}) \leq D(t_{10}, t_{11}) \leq ... \leq D(t_{n0}, t_{n1})
\]

\[
\Rightarrow \sigma_0 D(t_{00}, t_{01}) \leq \sigma_0 D(t_{10}, t_{11}) \leq ... \leq \sigma_0 D(t_{n0}, t_{n1})
\]

\[
\Rightarrow D(t_{00}, t_{01})^* \leq D(t_{10}, t_{11})^* \leq ... \leq D(t_{n0}, t_{n1})^*
\]

\(\square\)

It can now easily be shown that scaling of a time series does not affect performance of any algorithms (BU, EIISR, PSO-SGA and SGA) with respect to both profit and time fitness.
3.2. TIME SERIES SIMULATION

PSO It is clear that PSO sequentially improves its initial global best solution $X_0$ by finding better solutions $X_1, X_2, \ldots$ until it finds its final solution $X_r$, where $r$ is the total number of iterations. Since PSO finds sequentially better solutions, this implies that $f_1(X_1) \leq f_1(X_2) \leq \ldots \leq f(X_r)$, where $X_i$ here refers to the global best in iteration $i$. From theorem 1, we know that the order of the solutions will not change if the time series is scaled. For that reason, any sequence of solutions that PSO goes through to find the final solution, stays the same (assuming the same “random” numbers are generated during the execution of the algorithm); hence, keeping all other factors constant, PSO will perform exactly the same in a given time frame after scaling a time series.

SGA Let $Y_{\text{seg}}$ be a segment of a time series $Y$. Let $y_0$ be the lowest point in $Y_{\text{seg}}$ and $y_n$ the highest point in $Y_{\text{seg}}$. Therefore, $y(t_0) \leq y(t_1) \leq \ldots \leq y(t_n)$. If $Y$ is scaled to be $Y^*$, then, from theorem 2, $y(t_0)^* \leq y(t_1)^* \leq \ldots \leq y(t_n)^*$. This means that $t_0$ is still the time index of the lowest point, and $t_n$ is still the time index of the highest point in $Y_{\text{seg}}$. Therefore, at every optimisation step in the SGA algorithm, which entails finding the highest or lowest point in a given segment of the time series, the same time indices will describe the points found. Because of this, the SGA algorithm will perform the same after scaling the time series it operates on.

BSA-BU Let $D(t_{00}, t_{01}) \leq D(t_{10}, t_{11}) \leq \ldots \leq D(t_{n0}, t_{n1})$ be a sequence of such differences as described in theorem 3. If the time series $Y$ is scaled with $\sigma_0$, then from theorem 3, the sequence of differences will keep the same order, i.e. $D(t_{00}, t_{01})^* \leq D(t_{10}, t_{11})^* \leq \ldots \leq D(t_{n0}, t_{n1})^*$. The BSA-BU method works by removing the smallest $D$ from such a sequence above in every iteration, i.e. BSA-BU removes $D_0$ from a sequence $D_0 \leq D_1 \leq D_2 \leq \ldots \leq D_n$ in every iteration. Since such a sequence keeps its order if the series gets scaled (i.e. the two time indices defining the lowest value, $D_0$, will be the same time indices after the series is scaled), BU will remove the same pair of points at every iteration. This will result in the same solution (same fitness value) in the same number of iterations (same time taken).

BSA-EIISR Recall that in the section on the EIISR algorithm in the previous chapter, the concept of the prominence of a time series observation was introduced: if a trough was lowest in a window of radius $n$, it was referred to as a trough of order $n$. From theorem 2 it follows that if the trough at time $t_i$ is of order $n$ in time series $Y$, then the trough at time $t_i$
will still be of order $n$ in the scaled time series $Y^\ast$. This is because if a point at time $t_j$ was higher/lower than the point at time $t_i$ in the original series, then the point at time $t_j$ is still higher/lower than the point at $t_i$ in the scaled time series, which theorem 2 proves. Since the EIISR method works by finding the troughs of the highest order, it is easy to see that EIISR will find the same troughs in the scaled time series $Y^\ast$ as in the original time series $Y$. Therefore scaling the time series also does not have an effect on the EIISR method’s performance.

We have shown that if $\mu = 0$, then changing $\sigma$ will have no effect on any of the algorithms’ performance that are compared in this paper. Therefore the volatility was kept constant at $\sigma = 1$, in addition to assuming zero drift, $\mu = 0$.

### 3.2.5 Generation of fBm

In order to generate the process above, the process $B^H = \{B^H_t, t > 0\}$ was first generated, after which it was substituted into equation (3.8) to obtain each simulated observation, $Y_t$. Each $B^H_t$ was generated using a functionality built into Wolfram Mathematica 10 that can generate fBm with Hurst exponent $H \in (0, 1)$.

The full source code may be found in appendix F. Seed values are also given, so that the reader may duplicate the time series that optimisation methods in this study were applied to.

### 3.3 S-Races

As discussed in the literature review, S-Race was used to separate efficient and inefficient methods with regard to the two criteria: profit fitness and computation time. S-Race is described in more detail in this section. Some preliminaries which are necessary to know or recall in order to understand S-Race are first discussed, after which S-Race itself is described.

Note that we will from now on often refer to an algorithm with a specific parameter setting as a model. For example, SGA(1) and SGA(5) refer to the same algorithm, but with two different population sizes; therefore, they are different models. Of course, SGA(1) and EIISR, being different algorithms, are also different models. This is mostly for convenience
in distinguishing between the same algorithm but with different parameter settings, since these will also be compared against each other.

### 3.3.1 Pareto dominance

The following definition was adapted from Talbi (2009, p. 311).

**Definition 3.6 (Pareto dominance).** An objective vector \( u = [u_1 \ u_2 \ \ldots \ u_l] \) is said to dominate \( v = [v_1 \ v_2 \ \ldots \ v_l] \) (denoted by \( u \succ v \)) if and only if no component of \( v \) is larger than the corresponding element of \( u \) and at least one component of \( u \) is strictly larger, that is

\[
\forall i \in \{1, \ldots, l\} : u_i \geq v_i \land \exists i \in \{1, \ldots, l\} : u_i > v_i
\]

This form of dominance is also known as Pareto dominance.

Following from the definition of Pareto dominance, the definition below of Pareto optimality is also adapted from Talbi (2009, p. 311).

**Definition 3.7 (Pareto optimality).** A solution \( C_{hj} \in S \) is said to be Pareto optimal if for every \( C_{ij} \in S \), \( G(C_{ij}) \) does not dominate \( G(C_{hj}) \), that is, \( G(C_{ij}) \nleq G(C_{hj}) \).

In our specific case, the vector \( G(C_{ij}) \) is defined as

\[
G(C_{ij}) = \begin{bmatrix} g_{profit}(C_{ij}) & g_{time}(C_{ij}) \end{bmatrix}
\]

therefore any execution \( C_{hj} \) of a model \( C_h \) such that no execution \( C_{ij} \) of any other model \( C_i \) can improve over \( C_{hj} \) in terms of higher profit fitness and lower computation time (which implies a higher value for \( g_{time}(C_{ij}) \), the time fitness) is Pareto optimal.

### 3.3.2 Statistical error

Assume we are doing a test on the hypothesis \( H_0 \) at significance level \( \alpha = 0.05 \). The \( p \)-value calculated in the test will indicate where in the distribution implied by \( H_0 \) the observations
we are doing the test on is sampled from. Depending on whether a one-tail or two-tail test is performed, if \( p \) falls within some outlying region of the distribution that makes up 5% of the distribution, the null hypothesis is rejected. This is because we then know that there is a less than 5% chance that the observations come from the distribution assumed under the null hypothesis, \( H_0 \). Consequently, there is at most a \( \alpha = 5\% \) chance that the null hypothesis will be rejected, while it is in fact true (which is considered an acceptable rate, as \( \alpha \) is set beforehand). This is referred to as a type I error: the chance that \( H_0 \) is incorrectly rejected.

On the other hand, there is also the possibility of failing to reject \( H_0 \), when it is indeed false. To calculate the probability of this event, one would need to know the distribution of the alternative hypothesis (this distribution is assumed to be unknown in this study). This type of error is referred to as a type II error.

To summarise:

- **Type I error**
  - The probability that a true null hypothesis is rejected (false positive). This is equal to \( \alpha \).

- **Type II error**
  - The probability that a false null hypothesis fails to be rejected.

### 3.3.3 Sign test

The sign test is a nonparametric procedure for testing preference between two populations as a whole (Downing and Clark 2003, p. 404). Its best use is in cases where only the \(<\), \(=\), and \(>\) relations between two variables \( x \) and \( y \) are of interest. Being a non-parametric test, no assumptions on the distribution of samples are made; however, it carries less statistical power than alternate parametric tests, such as the t-test (Lehmann and Romano 2005, p. 538).

Assume we want to test for consistent differences between variables \( X \) and \( Y \). In particular, we wish to determine whether variable \( X \) is more likely to dominate variable \( Y \), than \( Y \) dominating \( X \), i.e. \( P(X > Y) > 0.5 \). Let \( n_{xy} \) be the number of cases where \( x > y \) and \( n_{yx} \) the number of cases where the opposite is true, and \( x \) and \( y \) are realisations of \( X \) and \( Y \).
3.3. S-RACES

The distribution of \( n_{xy} \) and \( n_{yx} \) may be modelled by a binomial distribution, \( \text{Binom}(s,p) \), where \( s = n_{xy} + n_{yx} \) and \( p \) is the probability of an occurrence of \( x > y \) (i.e. of \( n_{xy} \) incrementing) or, more concisely, \( p = P(X > Y) \). Since only the hypothesis \( p > 0.5 \) is of interest, it is not necessary to run the hypothesis test when \( n_{yx} \geq n_{xy} \), since then it is obvious that \( p \neq 0.5 \), considering the available evidence, \( n_{xy} \) and \( n_{yx} \). Since the hypothesis test is run only when \( n_{xy} > n_{yx} \), the only possibilities for \( p \) are \( p > 0.5 \) or \( p = 0.5 \); therefore the null hypothesis is \( H_0 : p = 0.5 \), and is tested against \( H_1 : p > 0.5 \).

Since the set of observed \( x > y \) and \( x < y \) is modelled as \( \text{Binom}(s,p) \), and the p-value is calculated under the assumption that the null hypothesis is true,

\[
\pi(n_{xy}, n_{yx}) = \frac{1}{2^s} \sum_{k=n_{xy}}^{s} \binom{s}{k}
\]  

(3.9)

where \( \pi(n_{xy}, n_{yx}) \) is the p-value for the one-tailed hypothesis test

\[ H_0 : p = 0.5 \]
\[ H_1 : p > 0.5 \]

Definition 3.8 (Inefficiency). Let \( n_{ab} \) be the amount of times that model B was Pareto dominated by model A. If, in every S-Race, there is some model A where the sign test rejects the null hypothesis with significance level \( \alpha \) and p-value \( \pi(n_{ab}, n_{ba}) \), then model B is referred to as inefficient.

Note that in definition 3.8 above, model A rejecting model B via S-Race does not necessarily mean that model A always dominates model B in the Pareto sense (definition 3.6) - instead, it means that there is sufficient statistical evidence to conclude (with a certain confidence) that A is more likely to dominate B than B dominating A in the Pareto sense.

3.3.4 Holm-Bonferroni method

When doing multiple comparisons, each with a critical value of \( \alpha \), the total probability of type I error (referred to as the family-wise error rate) increases if the \( \alpha \) value is not adjusted accordingly. There are various methods to make this adjustment, one of the simplest being
the Bonferroni correction. This approach divides the specified $\alpha$ value by the number of comparisons, $M$, to obtain a new significance level $\frac{\alpha}{M}$ (Frane 2015, pp. 13 - 14).

There exists, however, universally more powerful methods than the Bonferroni correction, such as the Holm-Bonferroni (Holm 1979, pp. 65 - 70) and Šidák (Šidák 1967, pp. 626 - 633) procedures. Attention will only be given to the Holm-Bonferroni method from here on, since this is the method that is used with S-Races in T. Zhang, Georgiopoulos, and Anagnostopoulos (2013, p. 1567).

The Holm-Bonferroni method, similar to the Bonferroni correction, is a procedure used to control the family-wise error rate (FWER) in a multiple comparison test. Algorithm 8, adapted from T. Zhang, Georgiopoulos, and Anagnostopoulos (2013, p. 1567), describes this procedure.

**Algorithm 8 Holm-Bonferroni procedure**

1: **procedure** HOMLBONFERRONI($\Pi$, $FWER$)
2: $\Pi$ is a set of the corresponding p-values of the $M$ hypotheses
3: $\alpha \leftarrow FWER$
4: Order $\Pi$ in ascending order as $\pi_1 \leq \pi_2 \leq ... \leq \pi_M$
5: Find the smallest $k = k_0$ such that $\pi_{k_0} > \alpha / (M + 1 - k_0)$
6: Reject the null hypotheses of the tests $\pi_k$ with $k < k_0$ $\triangleright$ If $k_0 = 1$, reject none, if no such $k$, then reject all
7: **return** the indices of rejected hypotheses, $i = 1, 2, ..., k_0$ if $k_0 > 1$, else **return** none
8: **end procedure**

### 3.3.5 Dominance matrix

It is very convenient to represent the number of times each model dominated, and was dominated by any other model, in the form of a matrix,

$$
\begin{bmatrix}
  n_{11} & n_{12} & \cdots & n_{1l} \\
  n_{21} & n_{22} & \cdots & n_{2l} \\
  \vdots & \vdots & \ddots & \vdots \\
  n_{l1} & n_{l2} & \cdots & n_{ll}
\end{bmatrix}
$$
where \( n_{ij} \) indicates how many times model \( i \) dominated model \( j \) and \( l = |C| \) (the number of models currently in the pool of models in S-Race). Thus, column \( i \) contains all the information on how many times, and by which model, model \( i \) was dominated. Similarly, row \( i \) contains all the information on how many times, and which model, model \( i \) dominated.

To find all the models that are sufficiently dominated by model \( i \) so that they can be eliminated from the pool of models, row and column values of each model are used in a series of sign tests. Assume we wish to find all models that model \( i \) dominates. Values from row and column \( i \) from the dominance matrix are then used in the sign tests on the \( i^{th} \) model.

Let

\[
\text{col}_i = [n_{i1}, n_{i2}, \ldots, n_{il}], \quad \text{row}_i = [n_{i1}, n_{i2}, \ldots, n_{il}]
\]

be the column and row values of model \( i \), respectively. Then, by doing a series of sign tests and obtaining p-values for \( \pi(n_{ij}, n_{ji}) \) for each \( j = 1, 2, \ldots, l \) where \( n_{ij} > n_{ji} \), a set of p-values is obtained as follows

\[
\Pi = \{\pi_1, \pi_2, \ldots, \pi_M\}
\]

where \( \Pi \) has the same meaning as in algorithms 8 and 9, where it is used as input for the Holm-Bonferroni procedure, \( \pi_j \) is the \( j^{th} \) calculated p-value and \( M \) is the number of models where \( n_{ij} > n_{ji} \). The output of the Holm-Bonferroni procedure then indicates which models were statistically significantly dominated by model \( i \). All those models are then eliminated from the pool of models, \( C \).

### 3.3.6 S-Race description and pseudocode

The list below summarises S-Race, after which it is described more formally in algorithm 9 on page 79. Both the list and the pseudocode are adapted from T. Zhang, Georgiopoulos, and Anagnostopoulos (2013, pp. 1567 - 1568).

- Start with a pool of models, called the candidate pool \( C \). Then proceed in iterations;
3.3. S-RACES

• in each iteration $j$, a new time series $Y$ is selected from a group of time series $J$. Each model in the candidate pool is tested on $Y$ (with the same value for $K$);

• compare the quantities $g_{\text{profit}}(C_{ij})$ and $g_{\text{time}}(C_{ij})$ resulting from running model $i$ in iteration $j$ on $Y$, against the corresponding values from all other models run on $Y$ in the same iteration $j$;

• for every model $i$, for every $k$ that satisfies $G(C_{ij}) \succ G(C_{kj})$, the quantity $n_{ik}$ is incremented by one;

• use the sign test, along with the Holm-Bonferroni method, to determine which models are inefficient. Reject inefficient models from $C$;

• if more than one model is in $C$ and not all time series in $J$ have been selected in previous iterations, go to step 2, otherwise, return final pool, $C$.

Definition 3.9 (Efficiency). Any model $A$ in the final pool at the end of at least one S-Race is referred to as efficient.

Note that when a model $A$ is efficient, according to definition 3.9 above, not all its executions are necessarily Pareto optimal (definition 3.7). Instead, it means that we have enough statistical evidence to conclude (with a certain confidence) that, in at least one S-Race, there exists no other model $B$, such that $B$ is more likely to dominate $A$ than $A$ is to dominate $B$ in the Pareto sense.

3.3.7 Type I error of S-Race algorithm

The following derivation for the link between the maximum probability of at least one type I error in the entire S-Race algorithm, $\Delta$, and the maximum FWER, $\alpha$, is adapted from T. Zhang, Georgiopoulos, and Anagnostopoulos (2013, p. 1568).

With the Holm-Bonferroni procedure, the type I error probability is limited within one family of paired comparisons by setting $\alpha$. In the S-Race procedure shown in algorithm 9, Holm-Bonferroni is applied at most $m - 1$ times within one iteration of the algorithm (T. Zhang, Georgiopoulos, and Anagnostopoulos 2013, p. 1568) where $m$ is the initial pool size (see appendix B, page 146). For each family of models tested, the FWER, controlled by the
Algorithm 9 S-Race algorithm

1: procedure SRace\((C, V, \Delta)\)
2: \(\alpha \leftarrow \text{AdjustAlpha}(\Delta)\)
3: \(C \leftarrow \{C_1, C_2, \ldots, C_m\} (m \geq 2)\)
4: \(J \leftarrow \) the validation set (a set of examples to run the models, \(C\) on)
5: while \(|C| > 1\) and \(|J| > 0\) do
6: Randomly sample a time series, \(Y\), from the validation set
7: \(J \leftarrow J \setminus \{Y\}\) \(\triangleright\) So that no time series may be sampled twice
8: Evaluate all remaining models on the time series \(Y\)
9: for each model \(C_i \in C\) do
10: \(\Pi \leftarrow \emptyset\)
11: for each model \(C_j \in C \setminus \{C_i\}\) do
12: \(\text{if } n_{ij} > n_{ji} \text{ then}\)
13: \(\Pi \leftarrow \Pi \cup \{\pi(n_{ij}, n_{ji})\}\) \(\triangleright\) Obtain sign test p-values for all models
14: \(\text{end if}\)
15: end for
16: if \(\Pi\) not empty then
17: \(\Gamma \leftarrow \text{HolmBonferroni}(\Pi, \alpha)\) \(\triangleright\) Get indices of all models to be rejected,
18: \(\text{and put them in a set } \Gamma\)
19: \(\text{end if}\)
20: for each \(k \in \Gamma\) do
21: \(C \leftarrow C \setminus \{C_k\}\)
22: end for
23: end while
24: Return \(C\)
25: end procedure
Holm-Bonferroni procedure (assuming that it cannot be applied more than $m - 1$ times in a single S-Race iteration) is given by

$$P(\text{At least one type I error in family}) \leq \alpha$$

In one iteration of S-Race, we have

$$P(\text{at least one type I error in iteration}) \leq (m - 1)\alpha \quad (3.10)$$

In equation (3.10) above, we use the fact that

$$P(A) \cup P(B) = P(A) + P(B) - (P(A) \cap P(B)) \Rightarrow P(A) \cup P(B) \leq P(A) + P(B) \quad (3.11)$$

Which makes equation (3.10) hold.

Letting $B$ be the maximum amount of iterations of the algorithm, and applying (3.11) in the same way we did to obtain (3.10), we have

$$P(\text{at least one type I error in S-Race}) \leq B(m - 1)\alpha$$

Letting $\Delta = P(\text{No type I errors in S-Race})$,

$$1 - \Delta \leq B(m - 1)\alpha$$

or

$$\Delta = P(\text{No type I errors in S-Race}) \geq \Delta_0 = 1 - B(m - 1)\alpha \quad (3.12)$$

Solving for the equality on the right,

$$\alpha = \frac{1 - \Delta_0}{B(m - 1)} \quad (3.13)$$
3.4. INVESTIGATION OF INPUTS

From equation (3.12), the true probability of making no type I errors in S-Race is \( \Delta \geq \Delta_0 \). If we therefore set the value for \( \alpha \) using \( \Delta_0 \), then \( \Delta_0 \) is the minimum confidence level for S-Race. For example, if \( \Delta_0 \) is set to 99%, then we can be at least 99% confident of not making any type I errors in S-Race. Equation (3.13) is represented by line 2 in algorithm 9.

Note that the bound for the occurrence of type I errors in the entire S-Race is very loose, since it assumes that no models are rejected during the entire S-Race. If models are rejected, which is very often the case in this study, then the true probability of no type I errors occurring increases even further.

3.4 Investigation of inputs

Aside from comparing models directly, the relative performance of the methods in the presence of different inputs will also be investigated. These inputs are discussed below.

3.4.1 Time series length

Since one of the main aims of this research is to investigate methods that can effectively perform preprocessing on large time series data sets, the series size is an obvious variable to investigate. This will show how well methods scale to time series size, i.e. how time series size affects the solution and time required for computation.

3.4.2 Number of buy-sell points

Increasing the number of buy-sell points to be found (up to a certain point) increases the complexity of the problem that each method must solve. It is also an essential parameter to be given by a practitioner using the buy-sell problem to analyse a given time series, since it determines the number of cycles found (or rate of compression). It is therefore of vital importance to investigate the performance of the algorithms under various values for this parameter.
3.4.3 Hurst exponent

The Hurst exponent, as explained in previous sections, has a very significant effect on the behaviour of the time series process. In particular, it has a direct effect on the jaggedness of a series, which influences how many local peaks and troughs there are. Since more peaks and troughs indicate more potential buy-sell points, investigating this parameter might give insight into which algorithms to apply for a series with specific characteristics.

3.5 Summary

In this chapter, details regarding the data that each algorithm was applied to, were given. Also, information on the methodology used for comparing the algorithms described in chapter 2 was given. Since the reader should now be familiar with this methodology, results of the comparisons between the different algorithms presented in the next chapter should be easier to understand, as well as the interpretation thereof.
Chapter 4

Results

4.1 Introduction

In this chapter, results and discussions on the performance of each of the methods described in chapter 2, are presented. It will become clear when reading this chapter that results from each section are dependent on results from a previous section. For this reason, results and discussion are combined into one chapter here.

The main purpose of this chapter is to find the best models to maximise the objective function described by equation (3.1). This would entail separating models that are efficient, from inefficient models (definitions 3.9 and 3.8 on pages 78 and 75, respectively).

Once we are left with a group of efficient models, a comparison of these models is done, since certain models in this group might perform better on one criterion than on the other criteria, and vice versa. In other words, some models in the final pool of efficient models might be better to use in certain situations over others in the final pool.

The optimisation problem that each model must solve is defined by three attributes: length of the time series ($Y_{size}$), number of buy-sell points ($K$), and Hurst exponent of the time series ($H$). Initially, models are compared in 9 distinct combinations of these three attributes. A combination of these three attributes will be referred to as a configuration. Each attribute is combined with all other attributes on two levels (high/low), with a midpoint configuration, which makes $2^3 + 1 = 9$ distinct configurations. It is therefore a two-level full factorial design (Mee 2009, p. 4) with a midpoint. Collecting data at the midpoint mainly
serves to provide a means to check the linearity of the effects between the two levels, the absence of which might provide sufficient motivation to run a full three-level experiment (Mee 2009, pp. 31-32). The configurations that make up the two-level design are shown in table 4.1.

S-Race is run on each configuration shown in table 4.1, which means that for each configuration, each model is run once on every time series generated for the configuration as long as it is still in the pool, as required by S-Race. For each S-Race, a total of 100 time series are generated, leading to a maximum of 100 iterations in every S-Race.

Different time series are generated for each configuration because of the stochastic nature of the time series, and of certain methods. In this way, S-Race accumulates statistical evidence on which models are inferior, and thus gives a summary of every model’s performance on each configuration by showing which models survived (were not rejected from) S-Race.

In each case, S-Race was run with a confidence level of at least $\Delta_0 = 99.9\%$ (see appendix B, page 146). The confidence level was set this high, since it was assumed that risking to keep an inferior model in S-Race (type II error) was better than risking to reject an efficient model, since in the former case, more information is accumulated on the model’s performance since it is executed more in the S-Race algorithm. The inferior model’s performance may afterwards be taken into account and one could say, for example, that it would have been rejected at another specified confidence level. This was thought to be better than not having any additional information on the model’s performance at all.

A graphical method that summarises the performance of each model in S-Race is provided throughout this chapter for most S-Races. This method is explained in section 4.1.2, below.

When results from the two-level experiment did not give satisfactory results (i.e. if it proved difficult to make conclusions from the experiment), S-Race was run on the additional configurations required to make the two-level experiment a three-level experiment in order to obtain more information for analysis. For the three-level experiment, each of the three attributes were combined on three levels: $H$ took on values 0.2, 0.5, 0.8; $Y_{size}$, 4000, 16000, 32000; $K$ assumed values 10, 50, and 100. The number of additional S-Races to be run in order to have a full three level experiment is 18; S-Race was not run again on the configurations in the two factor experiment.

Note that the reason only relatively large time series are chosen, is because this study is
4.1. INTRODUCTION

<table>
<thead>
<tr>
<th>Time series size ($Y_{size}$)</th>
<th>$H$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>0.2</td>
<td>10</td>
</tr>
<tr>
<td>4000</td>
<td>0.2</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>0.8</td>
<td>10</td>
</tr>
<tr>
<td>4000</td>
<td>0.8</td>
<td>100</td>
</tr>
<tr>
<td>32000</td>
<td>0.2</td>
<td>10</td>
</tr>
<tr>
<td>32000</td>
<td>0.2</td>
<td>100</td>
</tr>
<tr>
<td>32000</td>
<td>0.8</td>
<td>10</td>
</tr>
<tr>
<td>32000</td>
<td>0.8</td>
<td>100</td>
</tr>
<tr>
<td>16000</td>
<td>0.5</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 4.1: Configurations that S-Race is initially run on (two-level factorial design)

specifically focused on assessing how these algorithms perform when applied to larger time series sizes. We therefore did not use time series below a size of 4000 observations (except in one specific case, which will be discussed later in the chapter).

4.1.1 Algorithms

The methods compared in this chapter may be divided into stochastic and deterministic groups: a deterministic method provides the same solution every time it is run on a given time series with the same configuration. A stochastic method will give different solutions every time it is run on the same time series in the same manner that Monte Carlo methods yield variable solutions when used to approximate mathematical expressions, for example. In this study, deterministic methods do not have any parameters, while the stochastic methods each have at least one parameter.

The results chapter is divided up into sections, which might be seen as rounds in a competition: we start by analysing all the deterministic methods, which are made up of BU, EIISR, HR-BU, HR-EIISR and HR-FE2. S-Race is run on all these methods together. After this, SGA and HR-SGA are run against the models that survived sufficient S-Races with the deterministic methods. Finally, PSO-SGA and HR-PSO-SGA are run against the models that have survived sufficient previous S-Races.
4.1. INTRODUCTION

The reason the S-Races are run separately against each class of models is due to the fact that some of the methods have parameters. Analysing the methods that do not have parameters is easy, since there is no need to choose which parameter values to use for these methods in S-Race. These are all the deterministic methods, which is why they all get analysed together. When a method has parameters, sufficient models of these methods have to be chosen, so that enough parameter values are analysed. This is why S-Race is run separately with SGA, HR-SGA, PSO-SGA and HR-PSO-SGA — so that the results on parameter values will be easier to analyse.

4.1.2 Baseline plots

In each iteration of S-Race, a different realisation of the time series with the given characteristics was used to run all the models on. This implied that objective function values would have differed for each S-Race iteration, and in particular, what was a “good” objective function value in one iteration of S-Race was not necessarily a “good” fitness value in the next iteration. In order to graphically represent results, fitness values were therefore normalised by dividing each model’s objective function value by one specific model’s fitness value. The computation time was also normalised in the same manner. The model that divided every other model’s profit and time fitness values by its own, will be referred to as the baseline model. A scatter plot of all the normalised values will be referred to as a baseline plot. The baseline plots make it easy to see how the baseline model performed relative to the other models in the pool, and to a lesser extent, how well other models performed relative to each other.

On the y-axis, we plot \( y_{ij} = \frac{g(C_{ij})}{g(C_{Bj})} \), where \( g(C_{ij}) \) is the fitness value in the \( j \)-th iteration of S-Race of the \( i \)-th model in the pool, and \( g(C_{Bj}) \) is the fitness of the baseline model in the \( j \)-th iteration of S-Race. Note that \( y_{ij} \) and \( x_{ij} \) are used here to refer to a pair of values in the Cartesian plane, not as time series observations. The higher the value for \( y_{ij} \), the better model \( i \) performed in terms of fitness value against the baseline model, and a value of 1 indicates equal fitness values.

On the x-axis, we plot \( x_{ij} = \frac{t(C_{Bj})}{t(C_{ij})} \). Therefore, the higher \( x_{ij} \) is, the quicker model \( i \) performed relative to the baseline model in iteration \( j \) of S-Race, where a value of 1 also indicates the time taken was equal.
4.1. INTRODUCTION

Figure 4.1: Example of a baseline plot, with EIISR as the baseline method

A new set of reference axes can therefore be drawn, with the origin at (1, 1). It was noted that it is often difficult to read the computation time values due to large differences. For this reason, values on the x-axis were plotted on a logarithmic scale in most cases. Therefore on the x-axis we have \( x_{ij} = \ln\left( \frac{t_{C_Bj}}{t_{C_{ij}}} \right) \). In these cases, the origin of the new set of reference axes is at (0, 1).

If an observation is plotted in the first quadrant (top right) of the reference axes (including the reference axes themselves), then the corresponding model outperformed the baseline model in computation time and fitness. This means that the model in question dominated the baseline model for that time series, and S-Race would have registered it as a “win” for that model and a “loss” for the baseline model.

An observation in the second quadrant (top left, without touching the axes) implies that the model in question attained a higher fitness value than the baseline model, but took longer to compute, and therefore there was no dominance in that instance.

An observation in the third quadrant (bottom left, including the reference axes) implies that the baseline model exceeded the model in question in terms of fitness and lower computation time, which means that S-Race would have registered this as a “win” for the baseline model and a “loss” for the plotted model.

Finally, an observation in the fourth quadrant (bottom right, excluding reference axes) means that the plotted model arrived at a lower fitness value but was quicker to compute,
therefore no dominance took place in that instance.

Note that in each of these baseline plots, pairs of models where neither of the algorithms in the pair is not the baseline model, cannot be reliably compared to each other. For example, in figure 4.1, some of the black dots are higher than some of the green dots, but this does not necessarily indicate instances where the HR-EIISR method got a higher fitness value than the BU method. The higher black dots could also be the result of the EIISR method getting a lower fitness value than normal for a particular time series, in which case the corresponding BU fitness value for that time series could still be higher than the HR-EIISR fitness. However, if all of the green dots were higher than the black dots, then it can confidently be said that the BU method exceeded the HR-EIISR method in terms of fitness values for all the time series examples in S-Race (as can be seen in most of the plots in figure 4.4 on page 91, for example).

The same logic as above, applies to the x-axis. For example, once again in figure 4.1 it is clear that the green dots are all to the right of all the red dots, which means that the BU method was quicker to compute than the HR-FE2 method in all instances. But once again, if overlap occurs, one cannot confidently say in how many of the instances the BU method, for example, had a lower computation time than another model. The only comparisons that can be made reliably in these baseline plots are against the baseline model.

4.2 Deterministic methods

A comparison of the EIISR, BU, HR-EIISR, HR-BU and HR-FE2 methods, which comprise all the deterministic methods in this study, will now be made. In the following section (section 4.2.1), baseline plots and tables are shown that summarise the results obtained by running S-Race on the above-mentioned methods. These results are discussed in section 4.2.2.
4.2.1 S-Race results: baseline plots and summary

Baseline plots are shown in figures 4.2 to 4.4, with the EIISR method as the baseline model. Where the figures are arranged in a 2x2 grid, the plots on the left and right are for Hurst exponent 0.2 and 0.8 respectively, and top and bottom plots are for time series sizes 4 000 and 32 000, respectively.

Figure 4.2: Baseline plots for S-Race with all deterministic methods, where $K = 10$
Figure 4.3: Baseline plot for S-Race with all deterministic methods, where $K = 50$
Figure 4.4: Baseline plots for S-Race with all deterministic methods, where $K = 100$
### Table 4.2: Final pool after S-Race for deterministic methods (two-level factorial design)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>$Y_{\text{size}}$</th>
<th>$H$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td>BU</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>4000</td>
<td>0.2</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td>BU</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>4000</td>
<td>0.2</td>
<td>100</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td>BU</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>4000</td>
<td>0.8</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>4000</td>
<td>0.8</td>
<td>100</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td>BU</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>32000</td>
<td>0.2</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>32000</td>
<td>0.2</td>
<td>100</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td>BU</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>32000</td>
<td>0.8</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>32000</td>
<td>0.8</td>
<td>100</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td>BU</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>16000</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Final pool</td>
<td>EIISR</td>
<td>BU</td>
<td></td>
</tr>
</tbody>
</table>
### Table 4.3: Final pool after S-Race for EIISR and BU (Additional configurations for three-level factorial design, where $K = 10$)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>$Y_{size}$</th>
<th>$H$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIISR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>4000</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIISR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>16000</td>
<td>0.2</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIISR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>16000</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIISR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>16000</td>
<td>0.8</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIISR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>32000</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIISR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BU</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.2.2 S-Race results: discussion

In every S-Race in the two-factor experiment, all HR methods were eliminated. In most baseline plots, it can be seen that the methods from the HR family were dominated by the EIISR method, as they mostly fall into the third (bottom left) quadrant defined by the blue lines. When they were not dominated, they had a “draw” with the EIISR method (second quadrant), so they never dominated the EIISR method, which explains why they were eliminated from the S-Races. Also note that the BU method most likely also dominated the HR-FE2 and HR-BU methods (red and cyan dots, respectively) in many cases, since they are plotted mostly to the lower left of the BU cluster, although there is some overlap.

In all but one of the cases where $K = 10$, the BU method was also eliminated. This may seem surprising, since very few of the green dots, which represent the BU method, seem to fall into the third quadrant. However, figure 4.5, which is a detailed view of the section where the green cluster intersects the blue horizontal line, shows that in many cases BU had the same objective function value as EIISR (and in one case a lower value), but took longer to obtain it, since it is to the left of the vertical blue line. There were 19 of these instances in total, where there was no improvement over EIISR’s fitness by BU while taking longer to finish, meaning EIISR dominated BU 19 times, while BU never dominated EIISR once. This caused S-Race to eventually reject the BU method.

![Figure 4.5: Results for EIISR and BU methods where fitness values were close to each other (i.e. the ratio on the y-axis is around 1). This is for the same configuration as in figure 4.2, lower right](image)

In the case where $K$ is higher (taking on values of 50 and 100), it can be seen from
table 4.2 that both EIISR and BU made it to the end of the race with every configuration. The plots in figures 4.3 and 4.4 also confirm this, where it is clear that the BU method always got a higher fitness value than EIISR, but still took longer to finish its computation.

From the evidence for higher as well as lower values for $K$, it is clear that there is a trade-off between BU and EIISR: BU often gets a higher fitness value than EIISR (more so in the case where $K$ is higher), but EIISR is faster in obtaining a solution in every case. The fact that BU performs better with higher $K$ is in agreement with the observation made in chapter 2 on page 42 with respect to BU’s efficiency with higher $K$, although $K = 100$ is still not high enough for BU to have a lower computation time than that of EIISR.

Regarding the effects of time series size and Hurst exponent on the relative performance between BU and EIISR, there does not seem to be any noticeable, consistent effects, judging from the models that survived S-Race as well as the baseline plots.

Since all the methods from the HR family were rejected in all the S-Races in the full two-level factorial design (including the midpoint), it may be concluded that the HR-FE2, HR-BU and HR-EIISR are completely dominated by either BU or EIISR, which means that there is no benefit in running them over EIISR or BU. These particular HR methods will therefore not be considered further in this study (although the stochastic HR methods, HR-SGA and HR-PSO, will still be analysed).

Since BU was not rejected in all cases where $K = 10$, and there is no clear pattern regarding the races it was rejected from where $K = 10$, the results from the two-level experiment does not give a conclusive result regarding relative performance between BU and EIISR. S-Race was therefore applied to these two methods for the remaining configurations that S-Race would have been run on had we done a full three-level factorial experiment.

In the additional configurations to make a full three-level experiment, both EIISR and BU survived S-Race in all cases except two. In both those cases where one was rejected, it was always BU, and it was every time for where $K = 10$. However, there were still three cases where $K = 10$ when BU was not rejected, as shown in table 4.3, where the surviving models from S-Race for $K = 10$ are shown.

There is therefore also no clear pattern in the results from the full three-level experiment. Table 4.4, however, shows the maximum confidence level for BU to be rejected in each of the four configurations, where $K = 10$, where it was not rejected in the full three-level factorial
Table 4.4: Rejection criteria in S-Race for BU, where $K = 10$, and assuming that EIISR and BU are the only models in S-Race. Note that the significance level $\alpha$ for each hypothesis test is then adjusted to $\frac{\alpha}{100}$, since only a single comparison is done in each of the 100 iterations of S-Race. The confidence levels in the last column are adjusted for the 100 iterations in S-Race, based on the adjusted significance level.

In all the configurations in table 4.4, $K = 10$. It is clear from that table that had S-Race been run with only EIISR and BU in all cases, with a confidence level of 95%, then BU would have been rejected where $K = 10$. This gives a somewhat clearer picture of EIISR’s improved performance against BU when $K$ is low. However, the rejection of BU is not as decisive as with the HR methods, since, as can be seen in all baseline plots, BU gets an equal or better fitness in the overwhelming majority of cases, even where $K = 10$. BU was therefore still used in the following S-Races with other methods, even where $K = 10$.

### 4.3 HR-SGA

In this section, the HR-SGA method is investigated. Recall that the hierarchical method uses another method to find two buy-sell points in a chosen time-series segment at each step.
In HR-SGA, the hierarchical method uses SGA, with $K = 2$, to find two buy-sell points in each iteration.

Since SGA can have its population size varied, HR-SGA was tested with different population sizes for the SGA model that gets applied in each iteration. The range of population sizes for SGA that HR-SGA was tested with, is a slightly smaller range than that used for the ordinary SGA algorithm (shown in section 4.4). This can be justified, however, because SGA is only applied for $K = 2$ at each step of HR-SGA. Larger population sizes than 50 were therefore deemed unnecessary, since this simplifies the optimisation problem for SGA significantly, which eliminates the need for large population sizes.

Note that HR-SGA, where SGA with population size $P_{\text{SGA}}$ was used, is denoted HR-SGA($P_{\text{SGA}}$). HR-SGA with population sizes 1, 5, 10, 15, 20, 25 and 50 was tested. S-Race with these population sizes for HR-SGA, was ran on the same configurations as in the two-level full factorial experiment that the deterministic methods was run on. Both EIISR and BU were included in each S-Race.

Within one execution of HR-SGA, the population size for the SGA model applied at each iteration was kept constant. An investigation into whether the population size of the SGA model should be adapted based on the size and Hurst exponent of the series segment being optimised, could be a topic for future research.

### 4.3.1 S-Race results and summary

S-Race rejected HR-SGA for each configuration, regardless of the population size used. Only one baseline plot is included for this result, since the other baseline plots are very similar. The other baseline plots are included in appendix D on page 161, if the reader desires to study them.

In figure 4.6, it can clearly be seen that even the HR-SGA method with the highest population size, HR-SGA(50), took most of the time longer than BU to finish computation, and also most likely had a lower fitness value in all cases. It can also be seen that there is little difference in fitness values between HR-SGA(50) and HR-SGA(15). From this evidence, it seems unlikely that increasing population size past 50 would have been beneficial to HR-SGA’s performance.
4.4 SGA

Figure 4.6: Baseline plot for S-Race with HR-SGA, where $K = 50$

Since HR-SGA is rejected in each S-Race, this method is not considered further in this study.

4.4 SGA

The performance of the SGA method is investigated in this section. This time, values of $P_{SGA}$ were varied from 1 to 150, since now SGA is applied to problems where $K \geq 10$, making the optimisation problem more complex. Note that, as mentioned in section 2.3.3, SGA with population size $n$ is denoted SGA($n$).
4.4.1 S-Race results: baseline plots and summary

The performance of each SGA model is illustrated in figures 4.7 to 4.9. A summary of which models survived each S-Race is given in table 4.5.

*Figure 4.7: Baseline plots for S-Race with SGA, where $K = 10$*
Figure 4.8: Baseline plot for S-Race with SGA, where $K = 50$
Figure 4.9: Baseline plots for S-Race with SGA, where $K = 100$
<table>
<thead>
<tr>
<th>Configuration</th>
<th>$Y_{size}$</th>
<th>$H$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4000</td>
<td>0.2</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1), SGA(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>0.2</td>
<td>100</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1), SGA(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>0.8</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1), SGA(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>0.8</td>
<td>100</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1), SGA(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>0.2</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1), SGA(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>0.2</td>
<td>100</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>0.8</td>
<td>10</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1), SGA(5), SGA(150)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>0.8</td>
<td>100</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1), SGA(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16000</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Final pool</td>
<td>SGA(1), SGA(5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Final pool after S-Race for SGA (two-level factorial design)
4.4.2 S-Race results: discussion

Due to its low computation time, SGA(1) survived every S-Race. It can be seen from the baseline plots and the surviving models in table 4.5, that lower values of $K$ have a favourable effect on the performance of SGA, since in all instances but one where $K = 10$, only SGA models survived S-Race. In one instance where $K = 10$, EIISR also survived S-Race, however, it was dominated a total of 19 times by SGA(5), while EIISR never dominated either SGA(1) or SGA(5). This indicates that EIISR would have been rejected with 99.9% confidence if SGA(5) and EIISR were the only models in S-Race. It can also be seen that S-Race rejected any population sizes above five for all configurations except one: where series size is 32000, $K = 10$ and $H = 0.8$, SGA with a population size of 150 survived. SGA(5) was rejected in only one instance.

In all cases where $K \geq 50$, both BU and EIISR survived S-Race without being dominated once by any of the SGA models, as can be seen in the baseline plots.

Examining the baseline plots with regards to time series size and Hurst exponent, time series size seems to have no effect on relative performance between different models. Higher Hurst exponent seems to increase the variability in the relative fitness value of all models, although it does not have any consistent effect on the survival of models in S-Race.

Since the results regarding SGA(5) and SGA(150) are anomalies (and to a lesser extent, EIISR’s survival), S-Race was run on the remaining configurations required to make this a three-level full factorial experiment. SGA(1) and SGA(5) were included in all additional S-Races, since these two models survived the majority of the S-Races in the two-level experiment. Population sizes 125, 150 and 175 when $H \geq 0.5$ and $K \leq 50$, were also included, since the configuration that SGA(150) survived, seems to suggest that $H$ must be high and $K$ must be low for SGA with high population sizes to perform well. EIISR and BU were included in all additional S-Races.

SGA(1) survived all additional S-Races. EIISR and BU were rejected in all additional S-Races where $K = 10$, and survived all those where $K \geq 50$ without being dominated once, confirming the belief that SGA outperforms BU and EIISR when $K = 10$. However, it must still be noted that EIISR did get a fitness value above that of SGA(1) and SGA(5) in most instances where $K = 10$, as can also be seen in the baseline plots from the two-factor experiment. EIISR’s rejection from S-Race by SGA(5) is therefore not very decisive. Once
again, this is due to the fact that EIISR cannot dominate SGA(5) or SGA(1) since their computation times are always lower than that of EIISR.

SGA(5) survived all S-Races in the full three-factor experiment except those shown in table 4.6. The configurations that SGA(5) did not survive seem rather random, but note that the number of times SGA(5) was dominated by SGA(1) in every race was at least ten. Since SGA(5) never dominated SGA(1), the p-value that S-Race calculates for this comparison is at most $\pi(10, 0) = \frac{1}{2}^{10}$. Assuming that SGA(1) and SGA(5) are the only models in S-Race, the maximum confidence level to reject SGA(5) with that p-value is about 90.2%. SGA(5) was often dominated more than ten times, resulting in even lower p-values; therefore having confidence levels of 95% or even 97.5% would have resulted in SGA(5)’s rejection in most instances. However, even though SGA(5) was often dominated by SGA(1), SGA(5) was still considered as an acceptable model to use, since it displays lower variance in fitness than SGA(1), while retaining a reasonably low computation time.

Note that, since SGA(1) is always the fastest model, it cannot be dominated, since no other model can get a lower computation time. This might imply that S-Race is giving this model an unfair advantage, since it could consistently get very bad fitness values, and still not be rejected. However, since SGA(1) sometimes causes SGA(5) to be rejected and often dominates it, it shows that its fitness sometimes matches or exceeds that of SGA(5). SGA(1) therefore often gets reasonable fitness values even though it spends so little time finding a solution.

Examining the baseline plots from the two-factor experiment, one can see that SGA(1)’s variance in its performance is sometimes much higher than SGA with higher population sizes. For example, it is clear in some of the baseline plots, especially figure 4.7, that SGA(1) can at times get very low fitness values, in one or two cases out of 100 even getting below 50% of EIISR’s fitness value where $H = 0.8$ and $K = 10$. The main reason for this was explained in section 2.3.3 on page 25. SGA with higher population sizes therefore have the effect of reducing the variance of the solution, i.e. the solution becomes more reliable at the cost of computation time.

For $K = 10$ and to a lesser extent $K = 50$, there is also a clear trend in the baseline plots of SGA with higher population sizes having clearly a higher mean fitness value, not just decreased variance in fitness value. This explains why SGA with population sizes over 100 survived some S-Races with lower $K$. 
SGA(125) was the only model with higher population sizes that survived at least one S-Race in the additional runs. Together with SGA(150)’s survival in the two-level experiment, the configurations in the three-level experiment that any population size over 100 survived are shown in table 4.7.

Clearly, when $K = 10$ and $H > 0.2$, large population sizes perform better. There also seems to be some interaction between series size and $H$, since only at the largest series size, 32 000, does SGA survive S-Race where $H = 0.8$, while in all other cases, $H = 0.5$. However, more extensive testing is needed to make more confident conclusions on these interactions, if they exist, which may be a possible topic for future research.

In all S-Races that follow, SGA(1) and SGA(5) were included along with EIISR and BU. SGA with population size above 100 was not included in additional S-Races, since it did not survive that many S-Races, and was also never the only survivor.

### Table 4.6: All configurations where SGA(5) did not survive S-Race

<table>
<thead>
<tr>
<th>$Y_{size}$</th>
<th>$H$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>0.2</td>
<td>50</td>
</tr>
<tr>
<td>4000</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>16000</td>
<td>0.2</td>
<td>50</td>
</tr>
<tr>
<td>16000</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>16000</td>
<td>0.8</td>
<td>100</td>
</tr>
<tr>
<td>32000</td>
<td>0.2</td>
<td>10</td>
</tr>
<tr>
<td>32000</td>
<td>0.8</td>
<td>50</td>
</tr>
</tbody>
</table>

### Table 4.7: All configurations where SGA with high population sizes survived S-Race

<table>
<thead>
<tr>
<th>$Y_{size}$</th>
<th>$H$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>16000</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>32000</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>32000</td>
<td>0.8</td>
<td>10</td>
</tr>
</tbody>
</table>
4.5 PSO parameter tuning

Specific information on how each parameter was determined for PSO is given in this section. Note that each parameter of the PSO algorithm was already discussed in section 2.6.2. Here, we give information on more specific values that we used for the parameters. The reason that this is in the results section is due to the fact that we specify upper and lower bounds for PSO’s computation time based on computation time taken by the other methods (especially the upper bounds). This is so that the upper and lower bounds for PSO computation time may be fair.

4.5.1 PSO computation time

In preliminary experiments it was found that PSO could at times take very long to obtain a solution (compared to other methods), which is why we set bounds on how long each PSO model may take to finish computation. In the next section, a discussion is made on how computation time for each PSO/PSO-SGA model was determined, after which details on how other parameters were set, are provided in section 4.5.2.

Upper and lower bounds

Upper bounds for PSO’s computation time parameter (denoted $T_{PSO}$, which we sample randomly; see section 4.5.2 below) were determined and are shown in table 4.8. This upper bound is denoted $T_{upper}(|\Theta|)$, and was set according to time taken by other methods, the series size ($Y_{size}$), $H$ and $K$. The last three values all affect the size of the domain in which PSO must search for an optimal solution — the reasoning behind using them to set computation time is that a larger domain requires more time to search for an optimal solution. The domain of the objective function is denoted as the set $\Theta$. The domain size can be found by calculating $|\Theta| = \binom{N^*}{K} = \frac{N^*}{K(N^* - K)}$, where $N^*$ is the number of peaks and troughs in the time series. $N^*$ is a function of series size as well as $H$, since high values of $H$ imply that there is less jaggedness in the series, leading to less peaks and troughs, and vice versa for lower $H$. Since only peaks and troughs are included in the domain (recall that optimal buy-sell points may only occur on troughs/peaks), higher $H$ therefore means a smaller number of time series observations are in the domain, and lower $H$
4.5. PSO PARAMETER TUNING

| $Y_{size}$ | $K$ | $H$ | BU Average time (seconds) | $T_{upper}(|\Theta|)$ (seconds) |
|------------|-----|-----|---------------------------|---------------------------------|
| 32000      | 100 | 0.2 | 3.0                       | 20                              |
| 32000      | 100 | 0.8 | 1.5                       | 16                              |
| 4000       | 100 | 0.2 | 0.3                       | 14                              |
| 4000       | 100 | 0.8 | 0.15                      | 12                              |
| 16000      | 50  | 0.5 | 1.1                       | 10                              |
| 32000      | 10  | 0.2 | 3.0                       | 6                               |
| 32000      | 10  | 0.8 | 1.5                       | 5                               |
| 4000       | 10  | 0.2 | 0.3                       | 4                               |
| 4000       | 10  | 0.8 | 0.15                      | 3                               |

Table 4.8: Maximum possible computation times for PSO, denoted $T_{upper}(|\Theta|)$, where $K \geq 10$ implies a higher number are in the domain. From empirical results, it can be shown (in the case of the simulated time series used in this study, generated using Mathematica 10) that when $H = 0.2$, about 61% of the time series observations are peaks or troughs; for $H = 0.5$, 50% of the time series are either peaks or troughs and when $H = 0.8$, about 32.5% of the time series are peaks or troughs.

A lower bound for $T_{PSO}$, was also determined, therefore $T_{lower}(K) \leq T_{PSO} \leq T_{upper}(|\Theta|)$. The lower bound depends only on $K$, in particular, only whether $K > 2$ or not. A lower bound is necessary, since if PSO has very little time to find a solution, then it most likely would go through too few iterations to find a solution, and in extreme cases, might not even finish the first iteration (especially when swarm size is very large). As mentioned before, PSO often took relatively long to find a decent solution, therefore it was reasonable to assume that when swarm sizes are larger, and when the optimisation problem is relatively complex (such as when $K \geq 10$) giving PSO less than 0.3 seconds to finish might be unreasonable. The lower bound was therefore put in place so that we might not waste sampled PSO models. For $K = 2$, swarm sizes were limited to 20 particles, therefore in that case, $T_{lower}(K) = 0.01$. For $K \geq 10$, the lower limit was set as $T_{lower}(K) = 0.3$. 
4.5. PSO PARAMETER TUNING

| $Y_{size}$ | $H$ | BU Average time (seconds) | $T_{upper}(|\Theta|)$ (seconds) |
|------------|-----|---------------------------|-------------------------------|
| 32000      | 0.2 | 3.0                       | 9                             |
| 32000      | 0.8 | 1.5                       | 6                             |
| 4000       | 0.2 | 0.3                       | 4                             |
| 4000       | 0.8 | 0.13                      | 3                             |
| 1000       | Any | 0.02 - 0.06               | 1                             |
| 100 and 500| Any | 0.0015 - 0.025            | 0.5                           |

Table 4.9: Maximum possible computation times for PSO, where $K = 2$

For $T_{upper}(|\Theta|)$, if $|\Theta|$ is higher, then $T_{upper}$ is also higher. However, extra time given to algorithms that have larger values of $|\Theta|$ is not proportional to $|\Theta|$. This is because between some configurations, the difference in $|\Theta|$ can be incredibly large: the value of $|\Theta|$ for the configuration at the bottom of table 4.8 is about $3.6 \times 10^{24}$. For the configuration immediately above (second from the bottom), $|\Theta| = 4.06 \times 10^{33}$, which is more than a million times larger than the configuration at the bottom. The configuration at the top has $|\Theta| = 9.28 \times 10^{270}$ possible solutions, which confirms that $|\Theta|$ grows very rapidly as time series size and $K$ increases. It was therefore impractical to limit the computation time as a function of $|\Theta|$, particularly as a linear function. For $K \geq 10$, the upper bound of the computation time, $T_{upper}(|\Theta|)$, was therefore increased by between one to four seconds as $|\Theta|$ gets larger, as shown in table 4.8. Maximum computation times for when $K = 2$, and for the range of series sizes that PSO will be applied to in that case, are shown in table 4.9.

While the specific limitations in time is not rigidly defined by a function and most likely biased against the configurations with larger $|\Theta|$ (since $|\Theta|$ increases much faster than the increased time given), $T_{upper}(|\Theta|)$ in each case is nevertheless at least twice as much as the time that BU (the method with the longest computation time among the efficient algorithms) took to find a solution, and in most cases at least five times as much. Compared to time taken by other methods, the maximum possible computation time for PSO is therefore still fairly generous.
4.5.2 All parameters

As with SGA, PSO had to be tested with a reasonable diversity of models, i.e. parameter values, in order to determine the best model for the optimisation problem. PSO has four parameters: swarm size, inertia, neighbourhood size, and computation time. Testing the parameters in a factorial or grid fashion is very difficult: if we take only three different values for each parameter, and test each value in combination with all values of the other parameters, \(3^4 = 81\) different PSO models would have to be tested. Furthermore, in the above-mentioned case, only three different values for each parameter are tested, and for parameters where there is no guideline as to what its value must be, testing only three different values may not be enough.

To alleviate the problems above, a Monte Carlo approach was used. This entailed sampling random values for each parameter to obtain a total of 50 PSO models, each having a different combination of parameter values. Each of the 50 models were obtained by sampling a value from an appropriate statistical distribution for each of its parameters. Two distributions were used: the normal and uniform distributions.

The rationale behind using the normal distribution is that a parameter may be randomly sampled while centered around an educated guess for it (the mean, \(\mu\)). The magnitude of deviations from the educated guess may also be controlled via the standard deviation, \(\sigma\). Most sampled values will be around the guess, while a few sampled values will be farther from it. In this way, most models are sampled so that their parameters are relatively close to the educated guesses, while a few sampled values will be farther from the guesses so that alternative parameter values may also be investigated.

With the uniform distribution, an exact range of values has to be specified for the parameter. Also, for each range, values that were less likely to be good parameter values would have been just as probable to be sampled as values that were more likely to be good values. This makes the uniform distribution a better option if it is not certain which parameter values might be better than others, and an even better option if there are also definite boundaries for the parameter values.

**Swarm size** As mentioned in section 2.6.2, swarm sizes were restricted to values between 5 to 99, inclusive, following recommendations given by Chen, Montgomery, and Bolufé-Röhler (2015, p. 525). Since there was no certainty on whether lower or higher swarm sizes will give
better performance, swarm size was sampled from $U(5, 100)$, where each sampled value was rounded down to the nearest integer, effectively making the continuous uniform distribution a discrete distribution with bounds $[5, 99]$. However, when $K = 2$, swarm sizes were sampled from the distribution $U(2, 21)$, rounding values down in the same way as above, therefore effectively sampling from a discrete uniform distribution with bounds $[2, 20]$, inclusive. This is due to the fact that, at $K = 2$, PSO might have very little time to finish its computation, and also, since the optimisation problem is less complex when $K = 2$, it was thought unnecessary to have larger swarm sizes.

**Computation time** Computation times were sampled from a uniform distribution $U(T_{\text{lower}}(K), T_{\text{upper}}(|\Theta|))$. This is because there was no educated guess of what a good maximum computation time was for PSO-SGA, given the lower and upper time limits $T_{\text{lower}}(K)$ and $T_{\text{upper}}(|\Theta|)$, respectively. Also, all other parameters, such as inertia and swarm size, for example, were sampled once for each of the 50 models, then used for all S-Races. However, the distribution for computation time differed in terms of the bounds of the uniform distribution for each configuration (i.e. each S-Race). Therefore, instead of resampling computation time for each configuration, computation time for each model was sampled once from some uniform distribution, then scaled so that computation times matched the distribution for each configuration. Assuming that a computation time $T_1$ is sampled from $U(l_1, u_1)$, in order to scale $T_1$ to $T_2$ such that it comes from $U(l_2, u_2)$, the following scaling, or transformation, is done

$$T_2 = \frac{T_1 - l_1}{u_1 - l_1} (u_2 - l_2) + l_2$$  \hspace{1cm} (4.1)

A short proof of equation (4.1) may be found in appendix C, on page 156.

**Inertia** Inertia was already discussed in section 2.6.2, where results from Bansal et al. (2011, pp. 635 - 638) were discussed, after which it was determined that 0.7 is a reasonable initial guess for the inertia value. $N(0.7, 0.125)$ was therefore used as the distribution for inertia. The standard deviation was kept reasonably large, so that a large range of inertia values could be tested, but not so large that inertia values above 1 were sampled regularly, which would have been a waste of samples.
Neighbourhood ratio  For the neighbourhood ratio, it was assumed that if each particle had 10% of the whole swarm in its neighbourhood, it would be reasonable. The standard deviation was set to 0.1, therefore we sampled from $N(0.1, 0.1)$. In this way, a relatively large variety of neighbourhood sizes could be sampled, and in rare cases, even some very large ones where each particle has 30% of the swarm in its neighbourhood. If a value of zero or below was sampled, it was set to global best PSO, since a neighbourhood ratio below zero has no meaning. The mean is only one standard deviation above zero, therefore, due to the characteristics of the normal distribution, on average in 16% of cases global best PSO would be used.

To simplify the sampling process, each parameter was sampled from the same distribution for every value of $K$ and series size used, except in the case of maximum computation time, which was already discussed. The distribution from which each parameter was sampled from, is shown in table 4.10.

### Table 4.10: Distributions for each parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbourhood ratio</td>
<td>$N(0.1, 0.1)$</td>
</tr>
<tr>
<td>Inertia</td>
<td>$N(0.7, 0.125)$</td>
</tr>
<tr>
<td>Swarm size</td>
<td>$U(5, 99)$</td>
</tr>
<tr>
<td>Maximum computation time</td>
<td>$U(T_{\text{lower}}(K), T_{\text{upper}}(</td>
</tr>
</tbody>
</table>

### 4.6 HR-PSO-SGA

Results on PSO-SGA, where $K = 2$, are given here and discussed. The reason we specifically test for $K = 2$ in a separate section is because PSO-SGA with $K = 2$ gets used in HR-PSO-SGA. Testing PSO-SGA with $K = 2$ is therefore an efficient way to test HR-PSO-SGA without running the whole algorithm. Also, at this stage it is known how HR-SGA, HR-BU, etc. performs already: all the HR methods we tested so far were outperformed by other methods. Therefore, if PSO-SGA is rejected from S-Race by either SGA, EIISR or BU if $K = 2$, then we can say with a high measure of confidence that HR-SGA, HR-EIISR or HR-BU will outperform HR-PSO-SGA, since it means then that each of the optimisation steps in the hierarchical method is done less effectively by PSO-SGA than by the other
methods. This provided an easy way to rule out HR-PSO-SGA without wasting too much computational resources on testing it.

For $K = 2$, 50 PSO-SGA models were sampled using the distributions discussed above. S-Race was run on a wide variety of time series sizes: 100, 500, 1000, 4000 and 32000. For each time series size, two time series with the corresponding size and with Hurst exponents 0.2 and 0.8 were generated, with Hurst exponent 0.5 and series size 4000 being used as a midpoint. S-Race was therefore run 11 times in total. The high variation of series sizes that we test for here, is due to the fact that PSO-SGA will get applied to many different series sizes if used in HR-PSO-SGA, since a series segment can be any size.

4.6.1 S-Race results and discussion

Figure 4.10 is an example of what most baseline plots looked like for PSO-SGA, where $K = 2$. This example is for series size 4000 and $H = 0.5$, the midpoint configuration. The other baseline plots for S-Race with this group of models, with $K = 2$, are in appendix D, on page 164. Note that there are many more dots on the plot that belong to the PSO-SGA group than those which belong to one of the other models. This is because all PSO-SGA models are plotted as the same colour. Since there were 50 PSO-SGA models in this S-Race, plotting each model a different colour would not have made the plot much clearer to read.

Note that all baseline plots that follow from here on, have BU instead of EIISR as the baseline method.
In all 11 S-Race instances, PSO-SGA was run against the BU, EIISR, SGA(1) and SGA(5) models. All PSO-SGA models were rejected decisively from each S-Race, always before even the 30\textsuperscript{th} S-Race iteration. It is clear that most PSO-SGA models’ results are plotted in the third quadrant, implying that they were eliminated by the baseline model, BU. The few PSO-SGA models that are plotted in the fourth quadrant, were clearly either to the left, or to the bottom left of BU, SGA(1) or SGA(5), implying that those PSO-SGA models were most likely dominated by either one of those three models in those iterations of S-Race.

Since HR-SGA was outperformed by other methods, it follows then that HR-PSO-SGA will also be outperformed, and therefore no hierarchical methods were considered further.

4.7 PSO-SGA

In this section, the S-Race results on the PSO-SGA method are shown and discussed.
4.7.1 S-Race results: baseline plots and summary

In every configuration in the two-level full factorial design, all PSO-SGA models were rejected. Baseline plots for S-Race with PSO are given in figures 4.11 to 4.13. Since every PSO-SGA model was rejected for every configuration, we can say with reasonable confidence that it should not be used for cycle dating.
4.7. PSO-SGA

Figure 4.11: Baseline plots for S-Race with PSO-SGA, where $K = 10$
Figure 4.12: Baseline plot for S-Race with PSO-SGA, where $K = 50$
Figure 4.13: Baseline plots for S-Race with PSO-SGA, where $K = 100$
4.7.2 S-Race results: discussion

In the baseline plots for $K = 10$, it is clear the BU outperformed PSO-SGA in the vast majority of cases. Where series size was 4000, all methods were faster than PSO-SGA. However, this is due to the lower limit on the computation time imposed on PSO-SGA, which was 0.3 seconds, and all methods other than PSO-SGA were clearly faster than 0.3 seconds where series size was 4000.

In the case of series size 32000, some PSO-SGA models were faster than BU. However, these models were still slower than EIISR, and did not seem to outperform EIISR in terms of profit fitness. This led to their rejection from S-Race.

Where $K = 50$ and $K = 100$, all fitness values for PSO-SGA are well below BU in all cases. Once again, certain models’ sampled values for maximum computation time had them finish faster than BU. However, those PSO-SGA models faster than BU, were still slower than EIISR and clearly could not outperform EIISR, SGA(1) and SGA(5) in terms of fitness values, explaining why all PSO-SGA models were eventually rejected from every S-Race.

Note that in the baseline plots there is no evidence of higher computation times correlating with higher fitness values. Therefore, from the evidence we have, it seems that increasing allowed computation time would not have benefited PSO-SGA significantly.

4.8 PSO

Results from S-Race for PSO are given below. Note that these results are from the same executions of PSO as that which was shown for PSO-SGA, except that it was before SGA adjusted PSO’s final solution.

Take into account though, that in PSO-SGA, the adjustment that SGA(1) does to PSO’s final solution takes almost a negligible amount of time. Since SGA(1) could only improve the solution, it implies that the solution obtained by PSO-SGA always has a fitness value higher than or equal to the solution obtained by PSO. Therefore, by implication, PSO was also effectively rejected by S-Race. The results presented in this section is therefore purely for additional insight, especially into the way the initial solution for the SGA method (given by PSO) affected the final solution returned by PSO-SGA.
4.8.1 S-Race results: baseline plots

![Baseline plots for S-Race with PSO, where K = 10](image)

*Figure 4.14: Baseline plots for S-Race with PSO, where K = 10*
Figure 4.15: Baseline plot for S-Race with PSO, where $K = 50$
Figure 4.16: Baseline plots for S-Race with PSO, where \( K = 100 \)
4.8.2 S-Race results: discussion

In the baseline plots for PSO, it can be seen that, in most configurations, there seems to be some correlation between fitness value and computation time. In particular, when computation times are low, fitness values also seem slightly lower. However, in the case where $K = 10$ and $H = 0.8$, the computation time does not seem to make any noticeable difference, most likely due to the fact that when $H = 0.8$, there are far fewer local maxima in the optimisation landscape than when $H = 0.2$. Together with the fact that $K = 10$ in these cases (implying a less complex optimisation problem), this makes it easier for the PSO algorithm to improve its solution, since it is less likely to get stuck in local maxima; therefore, in those cases, even with low computation time, PSO was often able to find a reasonably good solution. In all cases where $K = 50$ and 100, the results suggest some correlation between longer computation time and higher fitness values.

This correlation of higher fitness values and longer computation time is not noticeable in PSO-SGA baseline plots. From this result, it seems unlikely that the optimality of the initial solution that the SGA algorithm obtained from PSO has a significant effect on the final result returned by PSO-SGA — however, a more rigorous analysis is needed to make definite conclusions on this.

4.9 Summary

The set of efficient models determined during this study were discussed. These were found to be the EIISR, BU and SGA methods. Some characteristics of the different algorithms were observed, as well as the effects that parameter values have on their performance. Note that, since the amount of raw data to produce the results given in this chapter is very large (with many results of executions of different models) the raw data is given on a CD, which was bundled with this dissertation. A short description of the S-Race data on this CD is given in appendix E.
4.9. SUMMARY

4.9.1 Hurst exponent and fitness value

It was observed in the results that the average fitness value for a given time series depended on its configuration. With two of the attributes that determine the configuration, time series size and number of buy-sell points, this is easy to understand: if there are more observations in the time series, keeping other factors constant, there are likely to be larger differences between given pairs of observations; if there are more buy-sell points, keeping other factors constant, then there are more pairs with positive vertical differences between their two observations, increasing fitness value. However, the Hurst exponent was also found to have a noticeable impact on fitness value. From the S-Race results in section 4.4, where SGA with various swarm sizes was pitted against BU and EIISR, and where $Y_{\text{size}} = 32000$ and $K = 10$, average fitness values for each model for every Hurst exponent tested are shown in table 4.11.

<table>
<thead>
<tr>
<th></th>
<th>Hurst</th>
<th>Average fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>BU</td>
<td>0.2</td>
<td>89.59</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>555.94</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>3996.60</td>
</tr>
<tr>
<td>EIISR</td>
<td>0.2</td>
<td>86.47</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>541.99</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>3913.61</td>
</tr>
<tr>
<td>SGA (5)</td>
<td>0.2</td>
<td>87.47</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>538.14</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>3566.88</td>
</tr>
</tbody>
</table>

Table 4.11: Average fitness values for BU, EIISR and SGA(5) in S-Race where $Y_{\text{size}} = 32000$ and $K = 10$

It is clear that a higher Hurst exponent implies higher fitness values on average. Higher fitness values imply higher possible returns, since the data in this study is always the logarithm of an underlying time series process (this was discussed in chapter 3, page 67). It therefore follows that the Hurst exponent of log prices could possibly indicate higher potential returns for a given asset. While there is heated debate regarding the relationship between risk and return (Malik 2015, pp. 25 - 40), including the Hurst exponent in this debate, due to its relationship to potential returns, is a possible topic for future research.
Chapter 5

Conclusions and recommendations

We present here our final conclusions made in light of the results observed in the previous chapter. For convenience, a table containing results of S-Race with all the efficient methods (the same as table 4.5 on page 102) is shown at the end of this chapter on page 128.

5.1 Efficient methods

Of all the methods tested, BU, EIISR and SGA were found to be efficient methods. Other algorithms (PSO, PSO-SGA and hierarchical methods) were found to be inferior to at least one of the efficient algorithms regardless of the configuration they were applied to.

5.2 Recommendations

Among the efficient methods, we were able to observe some patterns regarding computation time, fitness value and configuration. This allows us to make some recommendations on their usage, depending on certain factors.
5.2. RECOMMENDATIONS

5.2.1 If computation time is important

SGA method with population size 5 or below was consistently the fastest; therefore, if the analysis must be done as speedily as possible due to practical considerations, SGA with $P_{\text{SGA}} \leq 5$ should be used. SGA with population sizes 1 to 5 forms a spectrum, where lower population sizes are always faster but frequently have lower fitness values, compared to higher population sizes, where the method is slightly slower but often has higher fitness values.

5.2.2 If profit fitness is important

If it is important that the highest fitness value possible be obtained, while computation time is less important, the BU method will offer the highest chance of satisfactory performance. There was also some indication that SGA with population size above 100 might give satisfactory performance regarding fitness, in certain cases.

5.2.3 Different preferences between fitness and time

For different weightings in importance given to fitness and time, there are methods that accommodate this trade-off. The EIISR method was often slightly outperformed by the BU method in fitness, but usually also had a slightly lower computation time. As mentioned above, SGA with population sizes 1 to 5 also forms a spectrum in this regard; therefore if, compared to BU, slightly lower computation is required and some loss in fitness is acceptable, then EIISR is a good choice. SGA(5) could be used if slightly lower computation time than EIISR, but slightly better fitness than SGA(1) is required.

5.2.4 Configuration: number of buy-sell points

The number of buy-sell points, $K$, had a significant effect on relative performance. In particular, with higher $K$, BU always outperformed other methods in terms of fitness, with EIISR also surviving all S-Races in this case. With lower $K$, SGA with population sizes 5 or lower almost always outperformed other methods (i.e. SGA(1) was almost always the only model not rejected from S-Race with lower $K$). Therefore if $K$ is higher, BU and EIISR are
better options, while if $K$ is lower, SGA is recommended.

5.2.5 Configuration: Hurst exponent

The Hurst exponent was found to have a very small effect on relative performance. However, there was some indication that SGA with population size around 150 performed better with higher $H$. Indeed, the only times it survived were when $H \geq 0.5$ and $K = 10$. However, since SGA(150) often takes almost the same computation time as BU, sometimes even more, it should only be used if fitness is more important than computation time, and with the aforementioned configuration types.

Higher Hurst exponent increased the variability of relative fitness values of all SGA models, however, this did not have any noticeable effect on the survival of models from S-Race. Also, as shown at the end of the previous chapter, higher Hurst exponent increased average fitness values of all methods by a very large proportion.

5.2.6 Configuration: series size

Time series size was found to have little effect on relative performance between methods. This parameter did, however, increase computation time in all cases. For this reason, it is possible that with time series that are exceptionally large (much larger than those used in this study) the SGA method with lower population sizes might become preferable due to its fast computation time (especially with lower $K$), since differences in computation times will most likely become very significant then. In this study, differences in computation time between efficient methods were never more than four seconds.

5.2.7 Stochastic vs deterministic

SGA is a stochastic method: it often yields a different solution when applied to the same time series and configuration. If it is required that the same solution must be obtained when the method is applied repeatedly to the same series and configuration, then either the BU or EIISR algorithm must be used.
5.2.8 Robustness

SGA with population size 1 survived every S-Race. We see this as an indication that, within the framework of the methodology used to evaluate the algorithms, SGA with a population size of 1 is the most robust model. In other words, regardless of the configuration the model is applied to, it is highly unlikely that it will be dominated in the Pareto sense by any other known method.

5.3 Topics for future research

Increasing the individual iteration parameter of SGA, which we kept constant in this study, might prove useful since we are able to show that the order in which buy-sell points are adjusted has an effect on the final solution (see page 152 in appendix B).

Increasing the population size or individual iteration of SGA implies that more instances of the core SGA algorithm are run. These instances are run independently of each other, therefore parallel computation might provide a means to decrease computation time for SGA significantly, without any sacrifice in profit fitness (see page 154 in appendix B).

Alternative means of measuring the performance of the multi-criteria optimisation we performed here might give additional insights. For example, it is possible to combine computation time and profit fitness in a mathematical expression to provide a single numerical value of an algorithm’s performance. This could provide the means for a user to adjust his/her preference between computation time and profit fitness.

We found here that there could often be large differences in fitness values between methods, even between the EIISR and BU methods, and especially between the SGA method and the former two algorithms when its population size is low. A study will be needed to assess the impact that different fitness values have on time series analysis using the dated cycles, since different fitness values between executions of the methods imply that there are differences in the cycles found by the algorithms.

Drift and volatility of the time series used here were kept constant at $\mu = 0$ and $\sigma = 1$. Allowing these parameters to vary might yield additional insights into the relative performance of the methods investigated here.
Although not directly related to comparisons between algorithms, there was some reasonable evidence that higher $H$ implies higher potential profit for an asset price. Further investigation into this, especially with regard to the risk-return trade-off, could possibly yield valuable insight.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>$Y_{size}$</th>
<th>$H$</th>
<th>$K$</th>
<th>Final pool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4000</td>
<td>0.2</td>
<td>10</td>
<td>SGA(1), SGA(5)</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>0.2</td>
<td>100</td>
<td>SGA(1), SGA(5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EIISR, BU</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>0.8</td>
<td>10</td>
<td>SGA(1), SGA(5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EIISR</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>0.8</td>
<td>100</td>
<td>SGA(1), SGA(5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BU, EIISR</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>0.2</td>
<td>10</td>
<td>SGA(1), SGA(5)</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>0.2</td>
<td>100</td>
<td>SGA(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BU, EIISR</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>0.8</td>
<td>10</td>
<td>SGA(1), SGA(5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SGA(150)</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32000</td>
<td>0.8</td>
<td>100</td>
<td>SGA(1), SGA(5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BU, EIISR</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16000</td>
<td>0.5</td>
<td>50</td>
<td>SGA(1), SGA(5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BU, EIISR</td>
</tr>
<tr>
<td>Final pool</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Final pool after S-Race with all the efficient methods (same as table 4.5)*
Bibliography


Appendices
Appendix A

Data structures used for algorithms

In this appendix, we assume that the reader has already read through chapter 2, the algorithms chapter. We will discuss here lower-level implementation details of each method.

A.1 Data structures

A.1.1 Dynamic array

A dynamic array is a data structure that holds any number of elements, and elements may be removed or added after the array’s creation. The diagram in figure A.1 illustrates the typical structure of a dynamic array.

The important thing to note here is that elements are stored contiguously in memory. This fact will have an important bearing on the performance of the add and remove operations on a dynamic array.

![Figure A.1: Representation of a dynamic array containing nine elements](image-url)
A.1. DATA STRUCTURES

Removal operations

Assume that we wish to remove element $a_3$ from the array shown in figure A.1. Now, once we have removed it, there is a gap between $a_2$ and $a_4$, so the array is not contiguous anymore. We therefore close the gap by moving each element to the right of the gap one block to its left, and then deleting the final element. This requires that we execute $n - 4$ move operations. Removal from a dynamic array therefore has $O(n)$ complexity.

Insert operations

Assume that we wish to insert an element between $a_3$ and $a_4$. Now, an element is appended to the array, after which we move each element to the right of $a_3$ one block to the right. We then insert the new element into the gap that now exists between $a_3$ and $a_4$. Once again, we performed $n - 4$ move operations, so insertion into a dynamic array is also an $O(n)$ operation.

Access on stored objects

Since objects are stored contiguously in memory, it is very easy to access any object in the array. We simply need a reference to the first object. Assume we wish to find the object at index 10. The dynamic array looks at the location of the first stored object at index zero, then adds 10 to its location to determine the location of the object at index 10. Accessing any object in a dynamic array, when its index is known, is therefore an $O(1)$ operation.

A.1.2 Singly linked list

A singly linked list is a data structure of objects, where each object contains a pointer to the next object in the list. Unlike a dynamic array, objects are not necessarily stored contiguously in memory, hence the reliance on pointers. A graphical representation of a linked list is as follows.

![Figure A.2: Representation of a singly linked list](image)

Page 136
Removal of elements

In the diagram above, assume that we wish to remove the element containing $a_2$. All that needs to be done is we reassign the pointer of the node behind $a_2$ (containing $a_1$) to point to the node containing $a_3$. Now $a_2$ is not part of the list anymore, since if we traverse the list, $a_2$ cannot be reached anymore. Removal from a linked list is therefore an $O(1)$ operation, as confirmed by Cormen et al. (2001, pg. 206).

![Diagram of removal from a linked list](image)

*Figure A.3: Illustration of removal from a linked list*

In order to facilitate the removal shown above in figure A.3, one would need access to the node containing $a_1$, so that the pointer from that node may be reassigned to point to the node containing $a_3$. Often though, one initially only have access to node $a_2$, and since the pointers between nodes only point to the next node, accessing node $a_1$ directly from node $a_2$ would be impossible. In that case, a possible solution is to copy the contents from node $a_3$ to node $a_2$. This destroys the contents of node $a_2$, but now we have duplicates of node $a_3$. The next step is to remove the third node (the original node containing $a_3$) in the normal way. In this way, the linked list has the same contents as in figure A.3, as intended:

![Diagram of removal from a linked list](image)

*Figure A.4: Illustration of removal from a linked list*

The downside to removing elements in this manner is that, even though the contents of the list are the same, a different physical node was removed from the one that was intended. This might have consequences if other data depended on the content $a_3$ not changing location. In figure A.4 above, the third node does still contain $a_3$, but it is not in the list anymore, and the memory containing the node that contains $a_3$ was likely freed, which means that it might be used for other purposes, and is not guaranteed to contain $a_3$ any longer.
A.1. DATA STRUCTURES

Insertion of elements

Now assume that we wish to insert an element between $a_1$ and $a_2$. The pointer that originates from $a_1$ and points to $a_2$ is now reassigned to point to the new node, and the new node points to $a_2$. This is an $O(1)$ operation, as stated by Cormen et al. (2001, p. 206).

![Figure A.5: Illustration of insertion in a linked list](image)

Random access

Assume that we wish to access the 10th element in the linked list. Since elements are not contiguous in memory, we cannot use the same exploitation for this as we do for the dynamic array. We therefore have to go through each node in the linked list, and count, until we reach the 10th element, which will require going through 10 elements (10 operations). Since we may wish to access the last element, which will require $n$ operations (assuming the list is size $n$), accessing a random element is therefore an $O(n)$ operation in the worst case (Cormen et al. 2001, p. 205).

A.1.3 Doubly linked list

This structure is very similar to the singly linked list. The only difference is that each node is not only connected to the node in front, but also the node behind.

![Figure A.6: Representation of a doubly linked list](image)

Random access, insertion and removal complexity are the same as for the singly linked
list. The biggest difference here is that from any node in the doubly linked list, one can access the elements behind and in front of the node faster and in a much simpler manner. This is because each node now has two pointers: one pointing to the node in front and the other to the node behind. Therefore if one wants to remove the second node, with only a reference to that node, then it is a simple matter to access the nodes behind and in front and assign their pointers to each other, as shown below.

\[ \begin{array}{cccccccc}
  a_1 & a_2 & a_3 & a_4 & a_5 & a_6 \\
\end{array} \]

*Figure A.7: Removal in a doubly linked list*

In this way, any data that depended on the location of the contents \( a_3 \) not changing, would not be affected, since \( a_3 \) did not change location.

### A.1.4 Min-heap

A min-heap is a data structure that can be graphically represented as shown in figure A.8.

\[ \begin{array}{cccccccc}
  20 & \\
  50 & 70 & \\
  55 & 75 & 72 & 80 & 70 & 90 & \\
\end{array} \]

*Figure A.8: Representation of a min-heap*

A node that has two nodes connected to it below is referred to as the parent, and the two
nodes below it, as its children. A min-heap satisfies the property that the value contained in parent A is always less than or equal to the values contained each of its children B and C. We therefore always have the lowest value at the root, R. Note that this does not imply that the whole heap is ordered, but it may be considered as partially ordered.

Also note that it is possible to have a max-heap, which conversely always has the largest element at the root R, which it achieves by requiring that a parent A always contains a larger (rather than smaller) value than any of its children B and C.

**Insertion**

It is easy to see that, since we represent the min-heap as a binary tree, the number of levels in a min-heap is $\log(n)$, rounded up to the nearest integer. When we insert an element into a min-heap, we insert it at the bottom, connecting it to one of the available parents. The new value is then compared to its parent and, if lower, swaps places with the parent. The comparison is then made with the new parent and also swapped if lower, until it is compared to a parent that has lower value or until it is assigned the root node (in which case it is the lowest value in the heap). Since there are approximately $\log(n)$ levels in a heap and the same number of operations are done at each level, insertion has $O(\log(n))$ complexity (Cormen et al. 2001, p. 129). This procedure of moving the newly inserted value up gradually is known as percolation up.

**Removal of node**

When we remove a node, we assign the node with the largest value (one of the nodes at the bottom) to the node and remove it from the bottom (effectively moving it). We then do the opposite of percolation up: percolation down. Here we compare the node that was just assigned the largest value in the heap with its two children nodes, and if its value is larger than either of them, we swap it with the smaller of the two children. After this it is compared and possibly swapped in the same way with its new children nodes, until it has two children nodes that are both larger than it. Since the same number of operations are made at each level, and there are approximately $\log(n)$ levels in total, removal of the root node also has a complexity of $O(\log(n))$ (Cormen et al. 2001, p. 129).

Note that the insertion and removal of a node is done in such a way that the heap retains
its lowest value at the root after the operation is done.

A.2 SGA

A.2.1 Checking for improved points

Assume that we have a peak, and we are searching for a higher point for the peak. Also assume that this peak is point \( i \) in the time series. We scan every peak between buy-sell points \( i - 1 \) and \( i + 1 \) for a better position for the sell point. Since we preprocess the time series such that we have only peaks and troughs, we scan every second point in the reduced series, starting at \( x^*_{i - 1} + 1 \) and ending at \( x^*_{i + 1} - 1 \) (inclusive), since we know that peaks and troughs alternate in the reduced series. The same holds for buy points, except that we then search every trough to find the lowest point in the searchable area.

A.2.2 Data structures

In the adjustment queue, the only operations that were done were extracting the element at the front of the queue and adding an element at the back. For this, we used a singly linked list with a reference to the front element as well as to the element at the back. This is because a linked list adds and removes elements in constant time, while a normal array would have taken longer to remove the element at the front, since it then needed to adjust all other elements.

If we had used a dynamic array, potentially many additional operations would have been needed for each removal from the front of the queue, in comparison to the linked list which requires minimal (constant) additional operations for a removal anywhere from the data structure. Since we also only work with the elements at the front and back of the list, we did not need to access random elements in the list, which made the choice of a linked list more obvious.
A.3 BSA-EIISR

A.3.1 Data structures

Recall that in the BSA-EIISR method, we have clearance sets that monotonically decrease in size with increasing iteration numbers. A clearance set at iteration \( i \) contains the time indices of the time series observations that are \( i \) clear points. Now, in each iteration, when we test each element of the clearance set on whether it should be in the next clearance set, the order in which we test these elements have no effect. Therefore, we may test the elements in order, from the first to the last in the set, which means we have no need of random access of elements in the clearance set. Since we will regularly remove elements from the clearance set at any location in the set, a singly linked list was again the data structure of choice for implementing the clearance sets.

A.4 BSA-BU

Since the BSA-BU method works by removing pairs of buy-sell points that have the lowest vertical distance between them, a data structure that efficiently can supply us with the smallest element would be a good choice. We therefore make use of the min-heap, described earlier in this appendix, to store information on all the pairs we have at any point in the algorithm. The min-heap will order itself on the absolute vertical difference between the points that make up each pair, so that the pair with the lowest vertical distance is always at the root. The use of a min-heap is also recommended by Keogh et al. (2004, p. 11) for the original bottom-up method that they describe (which has some small, but significant differences with BSA-BU).

The min-heap, however, is only effective as a representation of the vertical differences in successive buy-sell points. Unlike the original bottom-up method, we also needed a representation of the relative positions of buy-sell points in the time series, which made it clear what the order of the buy-sell points are. The choice of data structure for this representation is a doubly linked list, as shown below.
A.4. BSA-BU

Figure A.9: Six buy-sell points during execution of BSA-BU

Figure A.10: Representation of the buy-sell points above via a doubly linked list

Each of the buy-sell points in figure A.9 is represented by a node in the doubly linked list in figure A.10. The order is also the same, i.e. the $i^{th}$ buy-sell point is represented by the $i^{th}$ node in the doubly linked list.

The min-heap for the buy-sell points above could be as shown in figure A.11
In the data structure in figure A.11, the top node of the min-heap will be removed next. This node corresponds to the difference between buy-sell points $a_3$ and $a_4$. Therefore, $a_3$ as well as $a_4$ will be removed from the linked list. However, the node containing $D_4$ in the min-heap also needs to be removed then, since it contains the difference between two buy-sell points, one of which does not exist anymore ($a_4$). $D_2$ also needs to be removed, since $a_3$, which does not exist anymore, forms one of the components of the pair which it represents. However, finding $D_2$ and $D_4$ will require in the worst case for us to search the whole min-heap, since they are not guaranteed to be at the top like $D_3$. Also, when we wish to remove the pair represented by $D_3$, for example, we would have to search the whole linked list in the worst case to find the buy-sell points representing $D_3$. Therefore, we have pointers from each of the elements in the min-heap to the first buy-sell in the pair it represents, and likewise, each node in the linked list (except the last) has a pointer back to its corresponding element in the min-heap. This eliminated the need to search for elements in the min-heap or the linked list as described above. This provided a significant speed up to the BSA-BU algorithm, since, if we apply it to a series that has 10 000 peaks and troughs, the linked list would contain 10 000 nodes and the min-heap 9 999. Searching for elements in such large data structures would have made the algorithm much slower than it currently is.
Similarly, the rationale behind using the linked list is because so many removals take place in arbitrary locations, which would have made it much slower had we used a dynamic array, looking at their relative complexities for removal. Also, since pointers in the heap depend on the corresponding nodes in the linked list keeping their physical location in memory, using a doubly, instead of a singly linked list was an obvious choice. This is due to the effects that removal from a singly linked list has on the element in front of the element to be removed (discussed earlier).
Appendix B

Additional notes

B.1 S-Race notes

B.1.1 Adjustment to $\alpha$

It was given in chapter 3 on page 61 that the value of $\alpha$ used for the Holm-Bonferroni procedure may be given by

$$\alpha = \frac{1 - \Delta_0}{B(m-1)} \quad (3.13)$$

The author’s main reasoning behind the value of $\alpha$ given above is that there could be no more than $m - 1$ executions of the Holm-Bonferroni method in a single iteration of S-Race, where $m - 1$ is the initial pool size. However, a simple counter example shows us that the Holm-Bonferroni method could indeed be run $m$ times, each time having a probability of type I error $\alpha$ at most.

Counter example

Suppose we have models A, B and C, and that there are two criteria, X and Y. Also, assume that these models are in reality equally good, i.e. none of them are inefficient. Any rejection that takes place in this S-Race is therefore a type I error. Now, suppose we have three iterations of S-Race as shown in table B.1.
Table B.1: Performance of each model in the example given here. Each pair of values corresponds to objective function values for two criteria. We assume each criteria must be maximised.

In table B.1, A dominates B in the first iteration, therefore $n_{AB} = 1$ while $n_{BA} = 0$ and model C has a draw with both A and B. In the second iteration, B dominates C, while A has a draw with both, therefore $n_{BC} = 1$ while $n_{CB} = 0$. In the third iteration, model C dominates model A, therefore $n_{CA} = 1$ while $n_{AC} = 0$. Therefore,

$$n_{AB} > n_{BA}$$

$$n_{BC} > n_{CB}$$

$$n_{CA} > n_{AC}$$

Therefore, there are three families of comparisons, so the Holm-Bonferroni method indeed is run three times and $m = 3$. This only occurs if no rejections take place for the first $m - 1$ applications of Holm-Bonferroni, since if A rejects B in the first test, then, since B is not in the pool anymore, the second test cannot be run. However, we will now show that this does not affect the confidence level significantly.

In the original proof for the connection between $\alpha$ and $\Delta_0$, the confidence level we specified for each S-Race in this study, we assumed the following

$$P(\text{at least one type I error in iteration}) \leq (m - 1)\alpha$$

Let us assume that the bound above is given by

$$P(\text{at least one type I error in iteration}) \leq m\alpha$$
B.1. S-RACE NOTES

Deterministic methods

<table>
<thead>
<tr>
<th>Initial pool size ((m))</th>
<th>Deterministic methods</th>
<th>SGA</th>
<th>PSO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>10</td>
<td>54</td>
</tr>
<tr>
<td>New confidence level ((\Delta_1))</td>
<td>99.875%</td>
<td>99.889%</td>
<td>99.888%</td>
</tr>
</tbody>
</table>

Table B.2: Actual minimum confidence levels for S-Races run in this study

That is, we now assume that there are \(m\) possible occurrences for type I error in an S-Race iteration, as we showed in the example above. It is easy to show that the adjustment for \(\alpha\) given by (3.13) now changes to

\[
\alpha = \frac{1 - \Delta_1}{Bm}
\]  

(B.1)

where \(\Delta_1\) is the confidence level we now assume. If we keep \(\alpha\) the same, and repeat S-Race with exactly the same performance by each model, then we will have exactly the same result, i.e. the same models will be rejected. Keeping \(\alpha\) the same, i.e. \(\alpha = \frac{1 - \Delta_0}{B(m-1)}\), we calculate the new overall confidence level of S-Race by substituting for \(\alpha\), then solving the following for \(\Delta_1\)

\[
\frac{1 - \Delta_1}{Bm} = \frac{1 - \Delta_0}{B(m-1)}
\]

which yields

\[
\Delta_1 = 1 - \frac{m}{m - 1} \Delta_0
\]

Since \(\Delta_0 = 0.999\) for all S-Races run in this study, \(\Delta_1\), the actual minimum confidence level, is dependent on how many models were in the initial pool, given by \(m\). In table B.2, we summarise what the actual minimum confidence levels were for all S-Races in this study. Since they change so little, and the bound for the actual confidence level is very loose, it is unlikely that this small change in the actual minimum confidence level would have affected results significantly. In the overwhelming majority of cases, the true probability of making no type I errors in S-Race was most likely above 99.9\% in any case, since in most S-Races, most models were rejected. We therefore still consider the results in this study valid.
B.2 BSA-EIISR notes

B.2.1 Clearance sets

In table B.3, we show many troughs are contained in each clearance set of the corresponding size (defined by the window radius). In figure B.1, this is illustrated graphically. It is clear that at lower window radii, the number of dots, or prominent troughs, decrease much quicker if we increase the window radius. For window radius 1, there are 270 prominent troughs, while for window radius 2, there are 135 prominent troughs, so a one unit decrease in window radius causes a difference of 135 elements between clearance sets $V_1$ and $V_2$. Between clearance sets $V_{20}$ and $V_{21}$, there is a difference of only one trough, and between clearance sets $V_{30}$ and $V_{33}$, there are no differences. This clearly highlights that at clearance sets $V_i$ where $i$ is higher, the difference between successive clearance sets become very small. Normally there are no differences between successive clearance sets with larger window radii, and when there are, usually only one element was not included in the next clearance set.

The stopping criterion of BSA-EIISR is such that, as soon as $|V_i| \leq \frac{K}{2}$, we stop optimisation. If there are very small differences between successive clearance sets $V_i$ and $V_{i+1}$, then it is more likely that $|V_F| = \frac{K}{2}$. In this case, we have exactly the amount of troughs we need, therefore there are no additional testing of a number of solutions to be done — we simply solve sub-problem 2, then return the solution. If it happens that $|V_F| < \frac{K}{2}$ the set $V_{extract} = V_i \setminus V_{i+1}$ will be very small, which means that the number of solutions we must test, given by $\left(\frac{|V_{extract}|}{\frac{K}{2} - |V_F|}\right)$, is also small.

<table>
<thead>
<tr>
<th>Window radius</th>
<th>No. of troughs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>270</td>
</tr>
<tr>
<td>2</td>
<td>135</td>
</tr>
<tr>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>30</td>
<td>7</td>
</tr>
<tr>
<td>33</td>
<td>7</td>
</tr>
<tr>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
</tr>
</tbody>
</table>

Table B.3: Number of prominent troughs in each window size
Figure B.1: Illustration of how number of troughs that are lowest in different window sizes varies

(a) Troughs with window radius 1

(b) Troughs with window radius 2

(c) Troughs with window radius 20

(d) Troughs with window radius 21
B.2. BSA-EIISR NOTES

(e) Troughs with window radius 30

(f) Troughs with window radius 33

(g) Troughs with window radius 40

(h) Troughs with window radius 50
B.3 SGA Notes

B.3.1 Order of adjustment

In SGA, one has to take note that the solution also depends on the order in which the points are adjusted. Consider the example shown in figures B.2 to B.4.

\[ K! \]

Figure B.2: Example of four points, indicated by dots, generated randomly on given time series

In figures B.3 and B.4, the red arrow shows the point that is adjusted first, the black arrow shows the point that is adjusted immediately after that point, and the red squares show the new positions of the points indicated by red dots. Notice how different the solutions are (shown in figures B.3b and B.4b) when we change the order in which we adjust the buy-sell points. It is clear that the solution in the second case is better than the solution in the first case. However, it is impossible to know (at this stage) which order of adjustment will yield an optimal solution, and it is also impractical to test every possible order if the number of buy-sell points $K$ is large (since we will have to evaluate $K!$ different orders to test every ordering). This is a shortcoming of the algorithm, but it could possibly be alleviated with more refinements to it. This could be a topic for future research.
B.3. SGA NOTES

(a) The arrows point to the horizontal position where the respective points will be adjusted to

(b) The final solution after the two adjustments were made. Note that the first black dot found the observation with smallest value in its accessible region, and the last black dot the highest, after we made the first two adjustments

Figure B.3: Illustration of how the solution is obtained if the first point is adjusted first

(a) The arrows point to the horizontal position where the respective points will be adjusted to

(b) The final solution after the two adjustments were made

Figure B.4: Illustration of how the solution is obtained if the second point is adjusted first
B.3.2 Parallel implementation

When we run SGA(n) (i.e. SGA with population size n), we effectively run SGA n times, each time generating a different initial solution, then solve it for each initial solution after which we choose the best solution from the n resulting solutions. It must be noted that the n executions of SGA that we make are independent of each other. No information is shared or needed between the different executions of SGA before all have finished execution.

This makes SGA very suitable for parallel implementation. A computer that has five CPU cores, for example, can in theory run SGA(5) for more or less the same time it takes to run SGA(1). No parallel implementations were done in this study; however, in a future study, this could be investigated.

B.4 PSO notes

B.4.1 Neighbourhood

Recall that the neighbourhood of each particle remains fixed, which means there is no relationship between how close two particles are to each other on the optimisation landscape and whether or not they are in a neighbourhood. The simplest way to define neighbourhoods in this case is to assign a unique number to each particle in the swarm. Number each particle 1 to $P_{PSO}$. If a particle is assigned number $n_i$, and neighbourhood size $n$ is even, then all particles from $n_l = n_i - \frac{n}{2}$ up to $n_r = n_i + \frac{n}{2}$ are in the neighbourhood of that particle. If $n_i - \frac{n}{2} \leq 0$, then $n_l = P_{PSO} + n_i - \frac{n}{2}$, and if $n_i + \frac{n}{2} > P_{PSO}$, then $n_r = n_i + \frac{n}{2} - P_{PSO}$. This ensures a ring-like structure as shown in figures 2.6c to 2.6e. A graphic explanation of this is shown in figures B.5 to B.7. Note that in each of these figures, the yellow blocks represent the particles in the neighbourhood of the particle represented by the red block.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
</table>

*Figure B.5: Illustration of the normal case, where neighbourhoods do not go over the “edge” of the numbering of particles*
B.4. PSO NOTES

Figure B.6: Illustration of the case where the leftmost index of the neighbourhood is \( n_l = P_{PSO} + n_i - \frac{n}{2} \)

Figure B.7: Illustration of the case where the rightmost index of the neighbourhood is \( n_r = n_i + \frac{n}{2} - P_{PSO} \)

Note that we will only have neighbourhood sizes that are even so that particles may have symmetric neighbourhoods around them. This makes it simpler to define neighbourhoods that are unique to each particle. Note that all particles will still “share” parts of their neighbourhoods with other particles (i.e. the neighbourhoods overlap, or to put it another way, each particle is part of more than one neighbourhood), but no two particles will have exactly the same neighbourhood of particles. By ensuring that each particle gets its information from a unique neighbourhood, we prevent any two particles from having, in every iteration, the same neighbourhood best, thereby improving exploration.

An alternative to the neighbourhood specified above is one where a particle’s neighbourhood of size \( n \) consists of the \( n \) particles that are closest to it in the optimisation landscape. However, this will increase computational cost considerably, especially if swarm sizes are high, since then each particle must do distance calculations with a large number of other particles in the swarm in order to determine its closest neighbours.
Appendix C

PSO model parameters

In the next two sections of this appendix, details on the 50 PSO models sampled for $K = 2$, and the 50 models sampled for $K \geq 10$, are shown. Note that where $K = 2$, the models are for where time series sizes are 100 and 500. The distribution that the computation time parameter was sampled from is therefore $U(0.001, 0.5)$. For $K \geq 10$, the models are from the pool for where $K = 10$, $H = 0.8$ and time series size is 4000. Those models’ computation time are therefore sampled from $U(0.3, 3.0)$.

If the computation time $T_2$ for a particular model, but for a different configuration than those specified above, is desired, then the transformation shown in equation (4.1) may be used:

$$T_2 = \frac{T_1 - l_1}{u_1 - l_1} (u_2 - l_2) + l_2$$

where $T_1$ is the computation time given for one of the configurations above, $u_1$ and $l_1$ are the upper and lower bounds for computation time of the given configuration above, and $u_2$ and $l_2$ are the upper and lower bounds for computation time of the configuration for which it is desired.

**Proof** We wish to transform a sample $T_1$ from the uniform distribution $U(l_1, u_1)$ to a sample $T_2$ from the uniform distribution $U(l_2, u_2)$. The cumulative distribution function (CDF) of the uniform distribution $U(a, b)$ is given by
\[ P(X \leq x) = F(x) = \frac{x - a}{b - a} \]

where \( x \) is a sample from the distribution. Now, since \( T_1 \) and \( T_2 \) must be from the same percentiles, we can set up the following equation

\[ P(X_1 \leq T_1) = P(X_2 \leq T_2) \Rightarrow \frac{T_1 - l_1}{u_1 - l_1} = \frac{T_2 - l_2}{u_2 - l_2} \]

solving for \( T_2 \), we obtain

\[ T_2 = \frac{T_1 - l_1}{u_1 - l_1} (u_2 - l_2) + l_2 \]
### C.1 PSO models where $K = 2$

<table>
<thead>
<tr>
<th>Swarm size</th>
<th>Inertia</th>
<th>Neighbourhood size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13</td>
<td>0.6635</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>0.778</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>0.8067</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.6423</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>0.7944</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>0.4429</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>0.479</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>15</td>
<td>0.5481</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>0.6976</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>0.8424</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>0.6166</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>0.6589</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>17</td>
<td>0.6675</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>18</td>
<td>0.78</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>0.8289</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>0.7506</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
<td>0.7027</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>17</td>
<td>0.7956</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>0.7132</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>0.8037</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>8</td>
<td>0.6124</td>
<td>2</td>
</tr>
<tr>
<td>23</td>
<td>5</td>
<td>0.7445</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>16</td>
<td>0.3767</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>0.6071</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>6</td>
<td>0.7435</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>12</td>
<td>0.6521</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>17</td>
<td>0.9173</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>5</td>
<td>0.6833</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
<td>0.8043</td>
<td>2</td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>0.6008</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>0.9445</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>6</td>
<td>0.6695</td>
<td>2</td>
</tr>
<tr>
<td>34</td>
<td>14</td>
<td>0.8108</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>11</td>
<td>0.8338</td>
<td>2</td>
</tr>
<tr>
<td>36</td>
<td>11</td>
<td>0.4797</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>19</td>
<td>0.7651</td>
<td>6</td>
</tr>
<tr>
<td>38</td>
<td>10</td>
<td>0.4866</td>
<td>2</td>
</tr>
<tr>
<td>39</td>
<td>17</td>
<td>0.6714</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>7</td>
<td>0.5286</td>
<td>2</td>
</tr>
<tr>
<td>41</td>
<td>13</td>
<td>0.9607</td>
<td>2</td>
</tr>
<tr>
<td>42</td>
<td>11</td>
<td>0.9107</td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td>8</td>
<td>0.7525</td>
<td>0</td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>0.8588</td>
<td>2</td>
</tr>
<tr>
<td>45</td>
<td>12</td>
<td>0.784</td>
<td>4</td>
</tr>
<tr>
<td>46</td>
<td>11</td>
<td>0.756</td>
<td>2</td>
</tr>
<tr>
<td>47</td>
<td>7</td>
<td>0.6908</td>
<td>2</td>
</tr>
<tr>
<td>48</td>
<td>19</td>
<td>0.5638</td>
<td>2</td>
</tr>
<tr>
<td>49</td>
<td>16</td>
<td>0.6032</td>
<td>2</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
<td>0.7035</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>14</td>
<td>0.6443</td>
<td>4</td>
</tr>
</tbody>
</table>

*Figure C.1: PSO model parameters where $K = 2$*
### C.2 PSO models where $K \geq 10$

<table>
<thead>
<tr>
<th>Swarm size</th>
<th>Inertia</th>
<th>Neighbourhood size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>14</td>
<td></td>
<td>0.3953</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td>6</td>
<td>0.4176</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>4</td>
<td>0.6407</td>
</tr>
<tr>
<td>5</td>
<td>94</td>
<td>8</td>
<td>0.6844</td>
</tr>
<tr>
<td>6</td>
<td>37</td>
<td>4</td>
<td>0.8622</td>
</tr>
<tr>
<td>7</td>
<td>39</td>
<td>2</td>
<td>0.7534</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>4</td>
<td>0.809</td>
</tr>
<tr>
<td>9</td>
<td>91</td>
<td>20</td>
<td>1.1744</td>
</tr>
<tr>
<td>10</td>
<td>18</td>
<td>2</td>
<td>0.8866</td>
</tr>
<tr>
<td>11</td>
<td>96</td>
<td>14</td>
<td>0.4187</td>
</tr>
<tr>
<td>12</td>
<td>46</td>
<td>2</td>
<td>0.5665</td>
</tr>
<tr>
<td>13</td>
<td>62</td>
<td>4</td>
<td>0.6239</td>
</tr>
<tr>
<td>14</td>
<td>88</td>
<td>6</td>
<td>0.923</td>
</tr>
<tr>
<td>15</td>
<td>23</td>
<td>2</td>
<td>1.8406</td>
</tr>
<tr>
<td>16</td>
<td>85</td>
<td>0</td>
<td>0.7386</td>
</tr>
<tr>
<td>17</td>
<td>90</td>
<td>18</td>
<td>1.6954</td>
</tr>
<tr>
<td>18</td>
<td>13</td>
<td>2</td>
<td>2.0653</td>
</tr>
<tr>
<td>19</td>
<td>60</td>
<td>2</td>
<td>2.1165</td>
</tr>
<tr>
<td>20</td>
<td>66</td>
<td>2</td>
<td>1.7229</td>
</tr>
<tr>
<td>21</td>
<td>66</td>
<td>10</td>
<td>1.0327</td>
</tr>
<tr>
<td>22</td>
<td>64</td>
<td>10</td>
<td>1.1393</td>
</tr>
<tr>
<td>23</td>
<td>56</td>
<td>0</td>
<td>1.1641</td>
</tr>
<tr>
<td>24</td>
<td>29</td>
<td>2</td>
<td>2.4738</td>
</tr>
<tr>
<td>25</td>
<td>66</td>
<td>6</td>
<td>1.9374</td>
</tr>
<tr>
<td>26</td>
<td>64</td>
<td>4</td>
<td>2.1555</td>
</tr>
<tr>
<td>27</td>
<td>44</td>
<td>6</td>
<td>1.3475</td>
</tr>
<tr>
<td>28</td>
<td>93</td>
<td>12</td>
<td>2.7755</td>
</tr>
<tr>
<td>29</td>
<td>55</td>
<td>10</td>
<td>1.5762</td>
</tr>
<tr>
<td>30</td>
<td>59</td>
<td>4</td>
<td>2.1533</td>
</tr>
<tr>
<td>31</td>
<td>57</td>
<td>0</td>
<td>1.7024</td>
</tr>
<tr>
<td>32</td>
<td>66</td>
<td>0</td>
<td>0.7556</td>
</tr>
<tr>
<td>33</td>
<td>98</td>
<td>20</td>
<td>1.7167</td>
</tr>
<tr>
<td>34</td>
<td>56</td>
<td>10</td>
<td>2.3413</td>
</tr>
<tr>
<td>35</td>
<td>41</td>
<td>4</td>
<td>2.512</td>
</tr>
<tr>
<td>36</td>
<td>61</td>
<td>10</td>
<td>2.9014</td>
</tr>
<tr>
<td>37</td>
<td>90</td>
<td>0</td>
<td>1.7982</td>
</tr>
<tr>
<td>38</td>
<td>43</td>
<td>12</td>
<td>1.0882</td>
</tr>
<tr>
<td>39</td>
<td>30</td>
<td>4</td>
<td>1.3383</td>
</tr>
<tr>
<td>40</td>
<td>93</td>
<td>14</td>
<td>2.0722</td>
</tr>
<tr>
<td>41</td>
<td>37</td>
<td>6</td>
<td>2.93</td>
</tr>
<tr>
<td>42</td>
<td>73</td>
<td>0</td>
<td>2.1455</td>
</tr>
<tr>
<td>43</td>
<td>70</td>
<td>0</td>
<td>1.7148</td>
</tr>
<tr>
<td>44</td>
<td>63</td>
<td>14</td>
<td>2.4314</td>
</tr>
<tr>
<td>45</td>
<td>36</td>
<td>0</td>
<td>2.5244</td>
</tr>
<tr>
<td>46</td>
<td>34</td>
<td>10</td>
<td>2.8008</td>
</tr>
<tr>
<td>47</td>
<td>75</td>
<td>20</td>
<td>2.3255</td>
</tr>
<tr>
<td>48</td>
<td>21</td>
<td>0</td>
<td>2.9055</td>
</tr>
<tr>
<td>49</td>
<td>39</td>
<td>0</td>
<td>2.9984</td>
</tr>
<tr>
<td>50</td>
<td>26</td>
<td>8</td>
<td>2.9054</td>
</tr>
<tr>
<td>51</td>
<td>29</td>
<td>4</td>
<td>2.9727</td>
</tr>
</tbody>
</table>

*Figure C.2: PSO model parameters where $K \geq 10$*
Appendix D

Additional baseline plots

D.1 HR-SGA baseline plots

Figure D.1: Baseline plots for S-Race with HR-SGA, where $K = 10$
Figure D.2: Baseline plot for S-Race with HR-SGA, where $K = 50$
Figure D.3: Baseline plots for S-Race with HR-SGA, where $K = 100$
D.2 PSO-SGA Baseline Plots (for $K = 2$)

Figure D.4: Baseline plots for S-Race with PSO-SGA, where $K = 2$ and time series sizes are large
D.2. PSO-SGA BASELINE PLOTS (FOR K = 2)

Figure D.5: Baseline plot for S-Race with PSO-SGA, where K = 2 and time series sizes are medium.
Figure D.6: Baseline plots for S-Race with PSO-SGA, where $K = 2$ and series sizes are small
Appendix E

Raw data description

The raw data for all S-Races executed in this study, is on the CD that came with this dissertation. The format of the data is discussed below.

For S-Race containing SGA and the deterministic methods, the columns were from left to right

- SRace iteration number
- Name of the method (which includes the population size for SGA methods)
- Fitness
- Computation time
- Winnings
- Losses
Figure E.1: Example of raw data for S-Race with deterministic methods

For S-Race containing PSO models, the columns were from left to right

- SRace iteration number
- Name of the method
- Swarm size
- Inertia
- Neighbourhood size
- Fitness

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>HR-EIISR</td>
<td>94.80247</td>
<td>1.2612121</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>BU</td>
<td>94.80247</td>
<td>3.069447</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>EIISR</td>
<td>91.56602</td>
<td>0.272250</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>HR-BU</td>
<td>94.80247</td>
<td>7.2125077</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>HR-FE2</td>
<td>94.80247</td>
<td>67.262190</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>HR-EIISR</td>
<td>81.27449</td>
<td>1.772345</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>BU</td>
<td>83.83888</td>
<td>3.0791877</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>EIISR</td>
<td>79.44155</td>
<td>0.2655260</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>HR-BU</td>
<td>81.27449</td>
<td>7.8310010</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>HR-FE2</td>
<td>81.27449</td>
<td>73.936560</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>BU</td>
<td>85.12876</td>
<td>3.0805650</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>EIISR</td>
<td>85.12876</td>
<td>0.2609360</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>HR-EIISR</td>
<td>83.13215</td>
<td>1.5383040</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>HR-BU</td>
<td>84.79017</td>
<td>10.33884</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>HR-FE2</td>
<td>84.79017</td>
<td>110.8890</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>EIISR</td>
<td>79.43671</td>
<td>0.2640630</td>
<td>7</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>BU</td>
<td>80.15233</td>
<td>3.1185140</td>
<td>8</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>HR-EIISR</td>
<td>77.81888</td>
<td>1.349921</td>
<td>5</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>HR-BU</td>
<td>78.14311</td>
<td>9.2583550</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>HR-FE2</td>
<td>78.14311</td>
<td>85.970330</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>BU</td>
<td>84.70183</td>
<td>3.1246110</td>
<td>10</td>
</tr>
<tr>
<td>22</td>
<td>5</td>
<td>EIISR</td>
<td>79.00034</td>
<td>0.2653320</td>
<td>8</td>
</tr>
<tr>
<td>23</td>
<td>5</td>
<td>HR-EIISR</td>
<td>76.84781</td>
<td>1.3531410</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td>HR-BU</td>
<td>79.09224</td>
<td>9.1529030</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>HR-FE2</td>
<td>79.09224</td>
<td>95.317120</td>
<td>0</td>
</tr>
</tbody>
</table>
• Time

• Winnings

• Losses

Where a column was not applicable to a method (e.g. BU, SGA and EIISR had no inertia) the models name was shown there instead.

The winnings and losses fields indicate how many times a model dominated any other models in the current pool, and how many times a model was dominated by any other models in the pool, respectively. Methods were sorted for each S-Race iteration according to their net winnings. This made it easier for a reader to see which models were doing better, especially with large pools.

It was not always the case that models with the highest net winnings survived S-Race, however. Most of the winnings might have come from that particular model A dominating many other weak models. Another model B which dominated model A, but which model A did not dominate back for some reason, might be responsible for its rejection from S-Race, while model B did not dominate many other models. For example, in figure E.2, it can clearly be seen that some PSO models, all of which were rejected, are ranked higher than SGA(5) and SGA(1), both of which survived that particular S-Race based on the raw data shown here. Those PSO models most likely got their winnings from dominating many other PSO models (of which there were 50 in total) but they clearly did not dominate BU, EIISR, SGA(5) or SGA(1) very often and were probably rejected by the BU method eventually.
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BU</td>
<td>BU</td>
<td>BU</td>
<td>BU</td>
<td>1193.84</td>
<td>0.139618</td>
<td>600</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>EISER</td>
<td>EISER</td>
<td>EISER</td>
<td>EISER</td>
<td>1193.723</td>
<td>0.020511</td>
<td>393</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>46</td>
<td>0.555332</td>
<td>2</td>
<td>1158.08</td>
<td>0.565503</td>
<td>395</td>
<td>40</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>96</td>
<td>0.677053</td>
<td>14</td>
<td>1158.07</td>
<td>0.418703</td>
<td>347</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>15</td>
<td>0.824534</td>
<td>4</td>
<td>1165.205</td>
<td>0.640699</td>
<td>345</td>
<td>55</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>37</td>
<td>0.662147</td>
<td>4</td>
<td>1171.101</td>
<td>0.862232</td>
<td>362</td>
<td>77</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>62</td>
<td>0.516624</td>
<td>4</td>
<td>1153.229</td>
<td>0.629075</td>
<td>320</td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>14</td>
<td>0.665235</td>
<td>4</td>
<td>1161.105</td>
<td>0.395318</td>
<td>271</td>
<td>26</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>23</td>
<td>0.854897</td>
<td>6</td>
<td>1151.629</td>
<td>0.417598</td>
<td>249</td>
<td>34</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>94</td>
<td>0.748438</td>
<td>8</td>
<td>1158.08</td>
<td>0.684411</td>
<td>277</td>
<td>67</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>39</td>
<td>0.933421</td>
<td>2</td>
<td>1168.438</td>
<td>0.753427</td>
<td>276</td>
<td>77</td>
</tr>
<tr>
<td>12</td>
<td>SGA(5)</td>
<td>SGA(5)</td>
<td>SGA(5)</td>
<td>SGA(5)</td>
<td>1173.754</td>
<td>0.006177</td>
<td>197</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>88</td>
<td>0.797389</td>
<td>6</td>
<td>1174.486</td>
<td>0.922582</td>
<td>291</td>
<td>102</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>91</td>
<td>0.753551</td>
<td>20</td>
<td>1151.629</td>
<td>1.174414</td>
<td>299</td>
<td>117</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>18</td>
<td>0.604955</td>
<td>2</td>
<td>1155.023</td>
<td>0.886635</td>
<td>251</td>
<td>100</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>85</td>
<td>0.408426</td>
<td>0</td>
<td>1145.508</td>
<td>0.738013</td>
<td>231</td>
<td>87</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>16</td>
<td>0.795164</td>
<td>4</td>
<td>1167.286</td>
<td>0.890967</td>
<td>223</td>
<td>94</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>66</td>
<td>0.807342</td>
<td>0</td>
<td>1135.856</td>
<td>0.755625</td>
<td>221</td>
<td>94</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>64</td>
<td>0.725288</td>
<td>10</td>
<td>1167.286</td>
<td>1.138961</td>
<td>258</td>
<td>132</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>56</td>
<td>0.762966</td>
<td>0</td>
<td>1140.657</td>
<td>1.161414</td>
<td>248</td>
<td>141</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>66</td>
<td>0.84246</td>
<td>10</td>
<td>1146.578</td>
<td>1.032555</td>
<td>217</td>
<td>133</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>90</td>
<td>0.736925</td>
<td>18</td>
<td>1158.08</td>
<td>1.69536</td>
<td>228</td>
<td>151</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>30</td>
<td>0.736129</td>
<td>4</td>
<td>1153.062</td>
<td>1.338264</td>
<td>217</td>
<td>149</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>66</td>
<td>0.656868</td>
<td>2</td>
<td>1165.205</td>
<td>1.729886</td>
<td>232</td>
<td>178</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>43</td>
<td>0.926628</td>
<td>12</td>
<td>1155.444</td>
<td>1.082427</td>
<td>181</td>
<td>136</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>55</td>
<td>0.407662</td>
<td>10</td>
<td>1158.08</td>
<td>1.576206</td>
<td>209</td>
<td>183</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>98</td>
<td>0.578717</td>
<td>20</td>
<td>1153.663</td>
<td>1.716658</td>
<td>212</td>
<td>198</td>
</tr>
<tr>
<td>12</td>
<td>SGA(1)</td>
<td>SGA(1)</td>
<td>SGA(1)</td>
<td>SGA(1)</td>
<td>1120.41</td>
<td>0.001472</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>44</td>
<td>1.028876</td>
<td>6</td>
<td>1136.906</td>
<td>1.347491</td>
<td>190</td>
<td>178</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>57</td>
<td>0.767331</td>
<td>0</td>
<td>1152.092</td>
<td>1.702425</td>
<td>202</td>
<td>192</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>13</td>
<td>0.614063</td>
<td>2</td>
<td>1161.508</td>
<td>2.06594</td>
<td>183</td>
<td>173</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>60</td>
<td>0.606697</td>
<td>2</td>
<td>1173.737</td>
<td>2.116526</td>
<td>161</td>
<td>185</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>23</td>
<td>0.545823</td>
<td>2</td>
<td>1148.993</td>
<td>1.840572</td>
<td>180</td>
<td>207</td>
</tr>
<tr>
<td>12</td>
<td>PSO</td>
<td>93</td>
<td>0.554106</td>
<td>14</td>
<td>1152.387</td>
<td>2.072164</td>
<td>176</td>
<td>204</td>
</tr>
</tbody>
</table>

Figure E.2: Example of raw data for S-Race with PSO
Appendix F

Source code

F.1 Mathematica 10 source code

This section contains code to generate the exact time series used in this study (using a specific seed value). The time series data is on the CD that accompanies this dissertation, however, this code could be used to generate it if the need arises.

The following source code was used to generate series sizes from 2000 to 32000, although only sizes from 4000 to 32000 were used. Note, in order to replicate the simulated time series data used in this study, run this code segment exactly as shown here. However, the output directory, (on the line that contains ‘‘c:/dating_main/data’’) may be modified.

In order to generate series sizes from 100 to 1000 (which included the series of sizes 100, 500, and 1000 used for the analysis of HR-PSO-SGA), assign the list \{100, 200, 500, 1000\} to the ‘‘lengths’’ variable on line 3 (instead of the list in the source code shown below).

```
SeedRandom[100];
exponents = Table[i, {i, 0.2, 0.8, 0.1}];
lengths = {2000, 4000, 8000, 16000, 32000};
umSeries = 100;

For[l = 1, l <= Length[lengths], l++,
   For[i = 1, i <= Length[exponents], i++,
      allSeries = {};
      For[numS = 1, numS <= numSeries, numS++,
```
F.2 Python source code

All source code that follows relates to the implementation of the algorithms in this study, as well as the tools for their comparison (i.e. S-Race).

F.2.1 Abstract base class for all methods

```python
import abc

class NoDataError(Exception):
    '''
    Exception defined for if data has not been sent to the algorithm class yet
    '''
    def __str__(self):
        pass
```
return "Data not set. First call set_data(series_obj) before calling optimise()"

class Optim(metaclass=abc.ABCMeta):
    """
    Abstract base class for all optimisation classes
    """
    def __init__(self, dim, long=None, trans_cost=0.0, from_hierarch=False):
        self.data_set = False
        self.from_hierarch = from_hierarch
        self.long = long
        self.trans_cost = trans_cost
        if dim % 2 == 0:
            self.dim = dim
        else:
            self.dim = dim + 1
        #self.pos = [0 for i in range(self.dim)]

def init_pop(self, new_dim=None):
    if new_dim is not None:
        self.dim = new_dim
    pass

@abc.abstractmethod
def _optimise(self, **kwargs):
    """Abstract optimisation function

    Abstract method. Derived classes' optimisation functions will implement this.
    """
    return

def optimise(self, **kwargs):
    if not self.data_set:
        raise NoDataError
    return self._optimise(**kwargs)

def set_data(self, series_obj):
    self.data = series_obj
    self.data_set = True
    if self.from_hierarch:
if self.long is None:
    self.long = self.data.begin_is_long()
    self.data.fix_long_short(self.long)
else:
    self.long = self.long
    self.data.fix_long_short(self.long)
self.series = [el.val for el in self.data.reduced]

def evaluate(self, attr="reduced"):
    value = 0.0
    for i in range(1, len(self.pos), 2):
        value += getattr(self.data, attr)[self.pos[int(i)]].val -
                 getattr(self.data, attr)[self.pos[int(i-1)]].val
    if self.long:
        self.fitness = value
        return value
    else:
        self.fitness = -value
        return -value

class IndivBase(metaclass=abc.ABCMeta):
    
    """Abstract base class for algorithms that implement populations of solutions""
    
    def __init__(self, parent):
        self.parent = parent
        self.data = parent.data
        self.trans_cost = parent.trans_cost
        self.dim = parent.dim
        self.long = parent.long
        self.pos = self.init_pos()
        self.val = self.evaluate()

    @abc.abstractmethod
    def init_pos(self):
        return

    def evaluate(self):
        total_trans_cost = 0.0
val = 0.0
if self.trans_cost > 0.0:
    for i in range(1, self.dim):
        total_trans_cost += self.data.reduced[self.pos[i]].val
    total_trans_cost *= self.trans_cost
for i in range(1, self.dim, 2):
    val += self.data.reduced[self.pos[i]].val - self.data.reduced[self.pos[i-1]].val
if not self.long:
    val *= -1
self.val = val
return val

F.2.2 FE2 method

from Algos import optim_virtual as ov

def gen_nums(num_lim, start):
    init = start
    while init < num_lim:
        yield init
        init += 2

class FE2(ov.Optim):
    """
    Implementation of FE2 algorithm, for use in hierarchical (HR) method only
    """
    Args:
    trans_cost (float): Transaction cost
    long (bool): True for long sequence of buy-sell points (where
    the sequence starts with a buy point and ends with a sell point)
    """
    data = None

    def __init__(self, trans_cost, long, *args, **kwargs):
        super().__init__(dim=2, long=long, \
                         trans_cost=trans_cost, from_hierarch=True)
```python
def __str__(self):
    return "FE2"

def get_name(self):
    return str(self)

def evaluate(self, _point1, _point2):
    return (_point2 - _point1)*self.up
            - self.trans_cost*(_point1 + _point2)

def _optimise(self):
    if self.long:
        self.up = 1
    else:
        self.up = -1
    self.points = [0, 1]
    self.val = self.evaluate(self.data.reduced\
                             [self.points[0]].val, self.data.reduced[self.points[1]].val)
    for i in gen_nums(len(self.data.reduced), 0):
        for j in gen_nums(len(self.data.reduced), i + 1):
            cand_val = self.evaluate(self.data.reduced[i].val,\
                                      self.data.reduced[j].val)
            if cand_val > self.val:
                self.points = [i, j]
                self.val = cand_val
    if (self.points[1] - self.points[0]) % 2 == 0:
        print(self.data.reduced[0].idx, self.data.reduced[1].idx)
    return self.points, self.val

F.2.3 Hierarchical method

import Algos.optim_virtual as optim_virtual
import copy

class Hierarch(optim_virtual.Optim):
    ""
    class structure that allows hierarchical method to take any optimisation method class
    as parameter and use it to do two dimensional optimisation in each 'region'.
    """
Args:
   dim (int): number of buy-sell points to extract
   trans_cost (float): desired transaction cost in optimisation procedure
   optim_method_cls (Any class derived from ov.Optim): a reference to the class that
   contains the
   optimisation method

```
def __init__(self, optim_method_cls, dim, trans_cost=0, long=True, **kwargs):
    self.optim_method_cls = optim_method_cls
    self.add_kwargs = kwargs
    super().__init__(dim, long, trans_cost, False)

def __str__(self):
    return "Hierarchical with " + self.optim_method_cls\
    (dim=2, trans_cost=self.trans_cost,\n    **self.add_kwargs).__str__()

def get_params(self):
    return {"Name": "HR-" + self.optim_method_cls\n    (dim=2, trans_cost=self.trans_cost,\n    from_hierarch=True, long=self.long,\n    **self.add_kwargs).get_name()}\n
def get_name(self):
    return self.get_params()["Name"]

def fix_end_regions(self):
    """Fix first and last buy-sell points

    fixes the first and last buy-sell points after
    optimisation is finished
    """
    lowest_val = float("inf")
    lowest_idx = -1
    for pos in range(0, self.pos[1], 2):
        if self.data.reduced[pos].val < lowest_val:
            lowest_idx = pos
            lowest_val = self.data.reduced[pos].val
```
highest_val = -float("inf")
highest_idx = -1
for pos in range(self.pos[-2] + 1, len(self.data.reduced), 2):
    if self.data.reduced[pos].val > highest_val:
        highest_idx = pos
        highest_val = self.data.reduced[pos].val

self.pos[0], self.pos[-1] = lowest_idx, highest_idx

def _optimise(self):
    self.pos = [0, len(self.data.reduced) - 1]
    self.region_valid = [[None, True]]
    self.num_reg = 1
    self.dat_attr = copy.deepcopy(self.data)

    iters = int((self.dim - 2)/2)
    pre_exit = False
    count = 0
    while len(self.pos) < self.dim:
        count += 1
        #if optimisation stopped due to an error
        if not self.step():
            pre_exit = True
            break
    if pre_exit:
        print("Optimization exited prematurely")
        self.fix_end_regions()
    return self.pos, self.evaluate()


def get_variation(self, region):
    """Get excess variation
    calculates and returns the excess variation for a specific region
    Args
    region (int): index of the region for which excess variation is required
    returns (float): excess variation of region
    """
if region >= self.num_reg:
    region = self.num_reg - 1
if self.region_valid[region][0] is not None:
    return self.region_valid[region][0]
the_sum = 0
end_idx = self.pos[region + 1]
for i in range(self.pos[region], end_idx):
    the_sum += abs(self.data.reduced[i+1].val - self.data.reduced[i].val)
self.region_valid[region][0] = the_sum - abs(self.data.reduced[self.pos[region]].val
- self.data.reduced[self.pos[region + 1]].val)
return self.region_valid[region][0]

def best_region(self):
    """find the region with the highest excess variation
    returns (tuple): (index of region, excess variation of region)
    """
    best_idx = None
    best_val = -1000000
    for i in range(0, self.num_reg):
        if (self.region_valid[i][1]):
            temp_val = self.get_variation(i)
            if (temp_val > best_val):
                best_idx = i
                best_val = temp_val
    return best_idx, best_val

def step(self):
    """A single iteration in hierarchical method
    In this method, searches are made for improved regions,
pairs of buy-sell points are added to the set of buy-sell points, etc.
    returns (bool): true if step was successful, false if an error occurred
    """
    best_reg, best_val = self.best_region()
    if best_val <= 0 or best_reg is None:
        return False
    else:
self.region_valid[best_reg] = [None, True]
if self.pos[best_reg + 1] - self.pos[best_reg] <= 1:
    return False
if self.optim_method_cls.__name__ == "EIISR":
    self.dat_attr = self.data.copy_subset(idx_start=self.pos[best_reg] + 1,
                                           idx_end=self.pos[best_reg + 1] + 1)
else:
    self.dat_attr = self.data.copy_subset(idx_start=self.pos[best_reg] + 1,
                                           idx_end=self.pos[best_reg + 1])
try:
    if best_reg == 0 or best_reg == len(self.region_valid) - 1:
        long = not self.long
        self.dat_attr.fix_long_short(self.long)
    else:
        long = not (self.dat_attr.orig[0].val < self.dat_attr.orig[-1].val)
    if self.optim_method_cls.__name__ == "PSORefined" or self.optim_method_cls.
        __name__ == "DERefined":
        self.add_kwargs["pop_size"] = max(int(len(self.dat_attr.reduced)/50), 4)
    optim2d = self.optim_method_cls(dim=2, trans_cost=self.trans_cost,
                                   from_hierarch=True, long=long, **self.add_kwargs)
    optim2d.set_data(self.dat_attr)
    optim2d.init_pop()
except IndexError:
    return False
optim_pos, optim_val = optim2d.optimise()
if optim_val <= 0.0:
    self.region_valid[best_reg][1] = False
    return True
if (optim_val < 0):
    return False
latest_insert = []
if self.dat_attr.begin_fixed:
    add_idx = 0
else:
    add_idx = 0
for pos, count in zip(optim_pos, range(len(optim_pos))):
    latest_insert.append(self.pos[best_reg] + 1 + pos + add_idx)
    self.pos.insert(count + 1 + best_reg, self.pos[best_reg] + 1 + pos + add_idx)
    self.region_valid.insert(count + 1 + best_reg, [None, True])
self.num_reg += 2
return True
F.2.4 SGA

from random import *
import time
import numpy
from Algos import optim_virtual as ov
from use_data import GenerateProper
import use_data as ud
import series
import copy
import numpy as np
from SGAQueue import SGAQueue

def np_sign(val):
    return numpy.sign(val)

class ParentError(Exception):
    def __init__(self, msg=None):
        self.msg = msg

    def __str__(self):
        if not isinstance(self.msg, str):
            return "Invalid parent for individual"
        else:
            return self.msg

class SmallSeriesException(Exception):
    ""
    Exception defined for if series size is too small relative to number of required buy-
    sell points (K). For example, if K is larger than the number of peaks/troughs
    ""
    def __str__(self):
        return "Dimension larger than series"

class SGA(ov.Optim):
    ""
    Class implementation of SGA method. Acts as a manager for all SGA individuals. All SGA
    individuals are implemented as objects of type Individual (defined below this
class)

Args:
  dim (int): Number of buy-sell points (K)
  pop_size (int): population size
  indiv_iter (int): individual iteration
  running_time (float): maximum running time in seconds.
  init_pos (list): initial position of buy-sell points
  long (bool): If true, do long optimisation (sequence starts with buy point, ends
  with sell point
  trans_cost (float): transaction cost
  from_hierarch (bool): True if the method is being implemented within hierarchical
  method as HR-SGA

""

def __init__(self, dim=0, pop_size=15, indiv_iter=1, \
        running_time=float("inf"), init_pos=None, long=True, \
        trans_cost=0.0, from_hierarch=False, **kwargs):
    self.pop_size = max(1, int(np.round(pop_size)))
    self.indiv_iter = max(1, int(np.round(indiv_iter))))
    self.total_pop_size = self.pop_size * self.indiv_iter
    self.initial_pos = init_pos
    self.init_pos = self.initial_pos
    if init_pos is not None:
        dim = len(init_pos)
    self.running_time = running_time
    super().__init__(dim, long, trans_cost, from_hierarch)

def __str__(self):
    return "SGA"

def get_name(self):
    return "SGA Pop: " + str(self.pop_size)

def get_params(self):
    return {"Name": self.get_name()}

def init_pop(self, new_dim=None):
    ""
    Initialise the SGA population.

    Also used as a way to reset the algorithm""
without creating a new object

Args:
    new_dim (int): if given (i.e. if new_dim is not None) reset the required number of buy-sell points to new_dim

Returns:
    None

```python
if new_dim is not None:
    self.dim = new_dim
self.pop = []
for i in range(self.pop_size):
    for j in range(self.indiv_iter):
        if j >= 1:
            if self.initial_pos is not None:
                self.pop.append(Individual(init_pos=self.initial_pos, parent=self))
            else:
                self.pop.append(Individual(init_pos=self.pop[-1].pos, parent=self))
        else:
            if self.initial_pos is not None:
                self.pop.append(Individual(init_pos=self.initial_pos, parent=self))
            else:
                self.pop.append(Individual(parent=self))

def max_indiv(self):
    """Find the individual with the highest fitness in the population"

    Returns (sga.Individual):
        Individual object with highest fitness
    """
    max_val = -1000000
    max_indiv = None
    for i in range(self.total_pop_size):
        if (self.pop[i].val > max_val):
            max_indiv = self.pop[i]
            max_val = self.pop[i].val
    return max_indiv
```
def _optimise(self):
    r"""Run algorithm to find optimal buy-sell points

    Args:
        None
    Returns (tuple):
        (position, fitness) of best individual
    r""
    begin_time = time.clock()
    for i in range(self.total_pop_size):
        if (time.clock() - begin_time) >= self.running_time:
            break
        self.pop[i].optimise()
    max_indiv = self.max_indiv()
    self.max_val = max_indiv.val
    self.fitness = self.max_val
    self.pos = list(max_indiv.pos)
    return max_indiv.pos, max_indiv.val

def evaluate(self):
    return self.max_val

class Individual(ov.IndivBase):
    r""
    Class implementation of a single individual in SGA method.
    
    Args:
        parent (SGA): a pointer to the SGA implementation whose population this individual belongs to

        other (Individual): serves as a copy constructor if this parameter is present

        init_pos (list): initial position for this individual
    r"
    
    def __init__(self, parent=None, other=None, init_pos=None):
        if other is not None:
            self.parent = other.parent
            self.pos = other.pos
            self.val = other.val
            self.dim = other.dim
self.long = other.long
self.trans_cost = other.trans_cost
self.data = other.data

elif init_pos is not None:
    self.parent = parent
    self.data = parent.data
    self.pos = copy.copy(init_pos)
    self.dim = len(init_pos)
    if not (self.dim == self.parent.dim):
        raise ParentError("individual's parent dimensions not equal to assigned
                        position's dimension."")
    self.up = 1
    self.long = parent.long
    self.initial_pos = copy.deepcopy(self.pos)
    gen_prop = GenerateProper(self, self.pos)
    self.pos = gen_prop.fix_pre_exist_pos()
    self.trans_cost = self.parent.trans_cost
    try:
        self.evaluate()
    except IndexError:
        raise IndexError(self.pos)
else:
    if not isinstance(parent, SGA):
        raise ParentError("parent argument not a valid type (must be of type 'SGA')")
    super().__init__(parent)
    self.initial_pos = copy.deepcopy(self.pos)
    gen_prop = GenerateProper(self, self.pos)
    self.pos = gen_prop.fix_pre_exist_pos()
    self.evaluate()

def init_pos(self):
    return list(sorted(np.random.randint(size=self.dim, low=0, high=len(self.parent.data.reduced))))

def adjust_one(self, pos_idx):
    """
    Adjusts a single buy-sell point in current solution.
    """

    Args:
    pos_idx (int): index (in vector of buy-sell points) of the point to adjust

Page 185
Returns (int):
    1 if point is moved forward, -1 if backward
    
orig_pos = self.pos[pos_idx]

mod = (pos_idx % 2)*2 - 1
if not self.long:
    mod *= -1

try:
    best_idx = self.pos[pos_idx]
    best_pt = self.parent.data.reduced[best_idx]
except IndexError:
    raise IndexError("self.pos[pos_idx]: ", self.pos[pos_idx], ",
    len(self.parent.data.reduced): ", len(self.parent.data.reduced))

initially, curr_idx refers to the lower bound of the region to be processed, and
next_idx refers to the upper bound of the region. after that, curr_idx keeps
track of the current index we are looking at and comparing

for count, sub_range in enumerate(self.search_list[pos_idx]):
    curr_idx = sub_range[0]
    next_idx = sub_range[1]

    while curr_idx < next_idx:
        if self.parent.data.reduced[curr_idx].val*mod >\
            best_pt.val*mod:
            best_idx = curr_idx
            best_pt = self.parent.data.reduced[best_idx]
        curr_idx += 2
    if count == 0:
        self.search_list[pos_idx][count][1] = \
        self.search_list[pos_idx][count][0] - 1
    else:
        self.search_list[pos_idx][count][0] =\n        self.search_list[pos_idx][count][1] + 1

self.pos[pos_idx] = best_idx
return numpy.sign(self.pos[pos_idx] - orig_pos)

def init_searchable_spaces(self):
    self.search_list = [None for i in range(self.dim)]
    for i in range(self.dim):
        if i == 0:
            self.search_list[i] = [[0, 0],
                                   [0, self.pos[i + 1]],
                                   [self.pos[i + 1] + 1, self.pos[i + 1]]]
        elif i == self.dim - 1:
            self.search_list[i] = [[self.pos[i - 1] + 1, self.pos[i - 1]],
                                   [self.pos[i - 1] + 1, len(self.data.reduced)],
                                   [len(self.data.reduced), len(self.data.reduced)]]
        else:
            self.search_list[i] = [[self.pos[i - 1] + 1, self.pos[i - 1]],
                                   [self.pos[i - 1] + 1, self.pos[i + 1]],
                                   [self.pos[i + 1] + 1, self.pos[i + 1]]]
    return self.search_list

def modify_search_list(self, idx, right):
    """Update search list for specific individual
    modifies the search list of the buy-sell point at index
    idx (in the list of buy-sell points).
    """
    if self.search_list[idx][1][0] < self.search_list[idx][1][1]:
        #if point at idx has not been modified yet
        if right:
            #if the point to the right of point at idx moved either away or towards it
            self.search_list[idx][1][1] = self.pos[idx + 1]
            self.search_list[idx][2][0] = self.pos[idx + 1] + 1
            self.search_list[idx][2][1] = self.pos[idx + 1]
        else:
            #if the point to the left of point at idx moved either away or towards it
            self.search_list[idx][1][0] = self.pos[idx - 1] + 1
self.search_list[idx][0][0] = self.pos[idx - 1] + 1
self.search_list[idx][0][1] = self.pos[idx - 1]

elif right:
    # if the point to the right of point at idx moved either away or towards it
    if self.pos[idx + 1] > self.search_list[idx][2][1]:
        self.search_list[idx][2][1] = self.pos[idx + 1]
else:
    # if the point to the left of point at idx moved either away or towards it
    if self.pos[idx - 1] + 1 < self.search_list[idx][0][0]:
        self.search_list[idx][0][0] = self.pos[idx - 1] + 1

def search_list_is_empty(self, idx):
    if self.search_list[idx][0][0] < self.search_list[idx][0][1]:
        return False
    if self.search_list[idx][1][0] < self.search_list[idx][1][1]:
        return False
    if self.search_list[idx][2][0] < self.search_list[idx][2][1]:
        return False
    return True

def adjust_all(self):
    """Adjust SGA individual solution, until finished
    calling this function causes the SGA individual to adjust points
    until its adjustment queue is empty
    returns None
    """
    count = 1
    adjust_idxs = set(list(range(self.dim)))
    self.init_searchable_spaces()
    adjustments = 0
    random_range = list(numpy.random.choice(list(adjust_idxs),
                                            len(adjust_idxs), replace=False))
    adj_queue = SGAQueue(random_range)
    while not adj_queue.is_empty():
        idx = adj_queue.root.idx
        dir = self.adjust_one(idx)
        adjustments += 1
if dir == -1:
    if idx < self.dim - 1:
        if self.search_list_is_empty(idx + 1):
            adj_queue.append_val(idx + 1)
            self.modify_search_list(idx + 1, False)
        if idx > 0:
            self.modify_search_list(idx - 1, True)
    
else:
    if idx < self.dim - 1:
        self.modify_search_list(idx + 1, False)
    if idx > 0:
        if self.search_list_is_empty(idx - 1):
            adj_queue.append_val(idx - 1)
            self.modify_search_list(idx - 1, True)
        if idx < self.dim - 1:
            self.modify_search_list(idx + 1, False)
    adj_queue.remove_front()

self.evaluate()

def optimise(self):
    
    """
    Wrapper function (decorator) for the adjust_all function.
    """
    
    returns (tuple): (final set of buy-sell points, final objective function value)
    """
    self.adjust_all()
    return self.pos, self.val

F.2.5  BSA-EIISR

from Algos import optim_virtual
from LinkedListEIISR import LinkedListEIISR
import copy

def perm_loop_help(lst, curr_idx, curr_num, total_num, all_els, all_perms):
    """
    Helper function for the permutations function defined below
    """
    if all_els is None:
        all_els = []
    for i in range(curr_idx, len(lst)):
all_els.append(lst[i])
if curr_num == total_num:
    all_perms.append(copy.deepcopy(all_els))
    del all_els[-1]
else:
    perm_loop_help(lst, i + 1, curr_num + 1,
                   total_num, all_els, all_perms)
    del all_els[-1]

def permutations(lst, num):
    all_perms = []
    perm_loop_help(lst, 0, 1, int(num), None, all_perms)
    return all_perms

class EIISR(optim_virtual.Optim):

    """
    Class implementation for EIISR algorithm.
    
    Args:
        dim (int): Number of buy-sell points, K
        long (bool): True if long sequence if buy-sell points is required
        trans_cost (float): transaction cost
    """

def __init__(self, dim, long=True, trans_cost=0.0,
              from_hierarch=False, with_troughs=True, **kwargs):
    self.with_troughs = with_troughs
    super().__init__(dim, long, trans_cost, from_hierarch)
    if with_troughs:
        self.troughs_mod = 1
    else:
        self.troughs_mod = -1

def get_name(self):
    return "EIISR"

def get_params(self):
    return {"Name": self.get_name()}

def set_data(self, series_obj):
super().set_data(series_obj)

def advance_next_round(self, idx, window_size):
    """Update clearance set with respect to observation at idx

    Determines whether a given time series observation is included in the next clearance set.

    Args:
    idx (int): time index of given time series observation
    window_size (int): window size which observation must be lowest in
    
    Returns:
    boolean (True or False): True if time series observation is included in next clearance set, False otherwise
    ""
    left_boundary = max(idx - window_size, 0)
    right_boundary = min(idx + window_size, self.data.orig_len - 1)
    if (self.data.orig[idx].val < self.data.orig[left_boundary].val)
    and (self.data.orig[idx].val < self.data.orig[right_boundary].val):
        return True
    if (self.data.orig[idx].val < self.data.orig[left_boundary].val)
    and (self.data.orig[idx].val == self.data.orig[right_boundary].val):
        return True
    if (idx == left_boundary) and (self.data.orig[idx].val < self.data.orig[right_boundary].val)
    or (idx == right_boundary) and (self.data.orig[idx].val < self.data.orig[left_boundary].val):
        return True
    return False

def find_best(self, begin_idx, end_idx):
    """
    Find highest (if self.troughs_mod = 1) or lowest
    (if self.troughs_mod = -1) time series observation
    between begin_idx and end_idx

    Args
    begin_idx (int): index defining beginning of region to be searched
    end_idx (int): index defining end of region to be searched
    
    returns (int): index of best point found
    """
best_idx = begin_idx
curr_idx = best_idx + 1
while curr_idx < end_idx:
    if self.data.orig[curr_idx].val * self.troughs_mod > self.data.orig[best_idx].
        val * self.troughs_mod:
        best_idx = curr_idx
        curr_idx += 1
return best_idx

def in_between(self, idxes):
    '''Find all peaks

    Finds all peaks between the troughs at idxes, and inserts them
    between the troughs in idxes

    Args
    idxes (list<int>): indexes of troughs already found
                        via the first sub-problem
    returns None
    '''
    insert_list = []
    for curr_idx in range(len(idxes)):
        if not self.forward:
            if curr_idx == 0:
                begin_idx = 0
            else:
                begin_idx = idxes[curr_idx - 1]
            end_idx = idxes[curr_idx]
        else:
            if curr_idx == len(idxes) - 1:
                end_idx = len(self.data.orig) - 1:
            else:
                end_idx = idxes[curr_idx + 1]
            begin_idx = idxes[curr_idx]
        insert_idx = self.find_best(begin_idx, end_idx)
        if self.forward:
            insert_list.append([curr_idx + 1, insert_idx])
        else:
            insert_list.append([curr_idx, insert_idx])
    insert_list.reverse()
    for insert_info in insert_list:
idxes.insert(insert_info[0], insert_info[1])

def all_permutations(self, prev_del_idxes, curr_idxes):
    num_idxes = self.dim/2
    required_num = num_idxes - len(curr_idxes)
    if required_num > 0:
        return permutations(prev_del_idxes, required_num)
    else:
        return [curr_idxes]

def _optimise(self, **kwargs):
    """Implements the abstract method _optimise.

    Args:
    None
    Returns (tuple):
    (position, fitness), where position and fitness are the buy-sell points and
    objective function value of the final solution found by the method
    """
    if self.long:
        self.forward = self.with_troughs
    else:
        self.forward = not self.with_troughs
    if self.forward:
        leave_out_start = 0
        leave_out_end = 1
    else:
        leave_out_start = 1
        leave_out_end = 0
    in_idxes = LinkedListEIISR()
    in_idxes.add_idx_range(leave_out_start, len(self.data.orig) - leave_out_end)
    window_size = 1
    remove_elements = []
    if self.dim == 2:
        condition_not_met = in_idxes.size > 5
    else:
        condition_not_met = in_idxes.size > self.dim/2
    while condition_not_met:
        remove_elements = []
        for in_idx in in_idxes:
if not self.advance_next_round(in_idx.idx, window_size):
    remove_elements.append(in_idx)
for el in remove_elements:
    in_idxes.remove_node(el)
window_size += 1
if self.dim == 2:
    condition_not_met = in_idxes.size > 1
else:
    condition_not_met = in_idxes.size > self.dim/2
in_idxes = list(in_idxes)
in_idxes = [node.idx for node in in_idxes]
prev_del_idxes = [node.idx for node in remove_elements]
top_fitness, top_pos = -float("inf"), None
if self.dim == 2:
    perm_func = permutations
    perm_args = prev_del_idxes + in_idxes, 1
else:
    perm_func = self.all_permutations
    perm_args = prev_del_idxes, in_idxes
for perm in perm_func(*perm_args):
    if len(perm) < self.dim/2:
        test_idxes = sorted(perm + in_idxes)
    else:
        test_idxes = sorted(perm)
    self.in_between(test_idxes)
    self.pos = test_idxes
    fitness = self.evaluate("orig")
    if fitness > top_fitness:
        top_fitness = fitness
        top_pos = copy.deepcopy(self.pos)
self.pos = top_pos
self.fitness = top_fitness
reduced_pos = find_reduced_idxes(self.data, self.pos)
return reduced_pos, self.fitness

def find_reduced_idxes(series_obj, idxes):
    reduced_idxes = []
curr_reduced_idx = 0
for count, pos_idx in enumerate(sorted(idxes)):
    while True:
        if series_obj.reduced[curr_reduced_idx].idx == \

series_obj.orig[pos_idx].idx:
    reduced_idxes.append(curr_reduced_idx)
    curr_reduced_idx += 1
    break
    curr_reduced_idx += 1
return reduced_idxes

F.2.6 BSA-BU

from Algos import optim_virtual
from LinkedList import LinkedList

class Node:
    
    """
    defines a node, which will be used to store a value in the heap
    actual values are stored in the linked list. the node accesses the
    element in the linked list via the link variable
    
    comparison operators are also defined for convenience
    """
    __slots__ = ["link"]

def __init__(self, link):
    self.link = link

def __le__(self, other):
    return self.link.val <= other

def __lt__(self, other):
    return self.link.val < other

def __ge__(self, other):
    return self.link.val >= other

def __gt__(self, other):
    return self.link.val > other

def __eq__(self, other):
    return self.link.val == other
```python
def __ne__(self, other):
    return not (self.link.val == other)

class BU(optim_virtual.Optim):
    
    """
    This class defines the BSA-BU method
    """

def __init__(self, dim, long=True, trans_cost=0.0, from_hierarch=False, **kwargs):
    self.ident = "BU"
    super().__init__(dim, long, trans_cost, from_hierarch)

def get_name(self):
    return "BU"

def get_params(self):
    return {"Name": self.get_name()}

def set_data(self, data):
    super().set_data(data)
    self.data.fix_long_short(long=self.long)
    self.series = self.data.reduced
    self.pos = LinkedList()
    for idx in range(len(self.series)):
        self.pos.push_back(idx)
    self.pos.assign_vals(self.series)
    if self.dim > len(self.series):
        self.dim = len(self.series)
    return
    self.heap_create()

def _optimise(self, **kwargs):
    while self.pos.size > self.dim:
        self.remove_lowest_el()
        self.pos = self.pos.make_pos_list()
    return self.pos, self.evaluate()

def heap_create(self, **kwargs):
    ordinary_pos_list = self.pos.make_ordinary_list()
    self.node_list = [Node(link=link) for link in ordinary_pos_list[:-1]]
```

---

Page 196
F.2. PYTHON SOURCE CODE

```python
self.heap = []
for node in self.node_list:
    self.heap_insert(node)
return self.heap

def heap_insert(self, val):
    """inserts a value into the heap

    Args
    val (float): value to be inserted
    returns None
    """
    self.heap.append(val)
    val.link.heap_idx = len(self.heap) - 1
    n = len(self.heap) - 1
    self.heap_bubble_up(n)

def heap_bubble_up(self, idx):
    """Adjust value at position idx

    called when an adjustment must be made. idx is the array index of the element
    that was removed. this function is called when the element that replaced
    the removed element at idx is smaller than its parent

    Args
    idx (int): index of element to be adjusted (in heap array)
    returns None
    """
    if idx > 0:
        parent_idx = (idx - 1)//2
        if self.heap[parent_idx] > self.heap[idx]:
            temp = self.heap[parent_idx]
            self.heap[parent_idx] = self.heap[idx]
            self.heap[idx] = temp
            self.heap[idx].link.heap_idx = idx
            self.heap[parent_idx].link.heap_idx = parent_idx
            idx = parent_idx
            self.heap_bubble_up(idx)

    def heap_bubble_down(self, idx):
```
"""Adjust value at position idx
called when an adjustment must be made. idx is the array index of the element
that was removed. this function is called when the element that replaced
the removed element at idx is larger than one of its children

Args
idx (int): index of element to be adjusted (in heap array)
returns None
"""

child1_idx, child2_idx = 2*idx + 1, 2*idx + 2
repl_value = float("inf")
repl_idx = None
try:
    if self.heap[child1_idx] < self.heap[idx]:
        repl_value = self.heap[child1_idx]
        repl_idx = child1_idx
except IndexError:
    pass
try:
    if self.heap[child2_idx] < self.heap[idx]:
        if self.heap[child2_idx] < repl_value:
            repl_value = self.heap[child2_idx]
            repl_idx = child2_idx
except IndexError:
    pass
if repl_idx is not None:
    temp = self.heap[repl_idx]
    self.heap[repl_idx] = self.heap[idx]
    self.heap[idx] = temp
    self.heap[repl_idx].link.heap_idx = repl_idx
    self.heap[idx].link.heap_idx = idx
    self.heap_bubble_down(repl_idx)

def heap_remove(self, idx):
    """Remove a specific element from the heap, identified by idx

Args
idx (int): the index of the element in the heap array
returns (Node): heap node that was removed
"""
ret_val = self.heap[idx]
self.heap[idx] = self.heap[-1]
self.heap[idx].link.heap_idx = idx
del self.heap[-1]
if idx < len(self.heap):
    if self.heap[idx] > ret_val:
        self.heap_bubble_down(idx)
    elif self.heap[idx] < ret_val:
        self.heap_bubble_up(idx)
return ret_val

def remove_lowest_el(self):
    """Remove lowest element from the heap (the root)

removes the lowest (topmost) element from the heap
and adjusts the heap accordingly
returns None
"""
rem_node = self.heap[0]
if rem_node.link.next is self.pos.back:
    self.pos.del_node(rem_node.link.next)
    self.heap_remove(rem_node.link.prev.heap_idx)
    self.heap_remove(rem_node.link.heap_idx)
    self.pos.del_node(rem_node.link)
elif rem_node.link is self.pos.root:
    self.heap_remove(rem_node.link.heap_idx)
    self.heap_remove(rem_node.link.next.heap_idx)
    self.pos.del_node(rem_node.link.next)
    self.pos.del_node(rem_node.link)
else:
    if rem_node.link is self.pos.back:
        print ("INVALID: rem_node.link is self.pos.back")
        try:
            self.heap_remove(rem_node.link.heap_idx)
            self.heap_remove(rem_node.link.next.heap_idx)
        except IndexError:
            while rem_node.link is not None:
                print (rem_node.link.idx)
                print (rem_node.link.next.heap_idx)
                print (len(self.heap))
rem_node = self.heap[rem_node.link.next.heap_idx]
raise IndexError
new_first_link = rem_node.link.prev
new_2nd_link = rem_node.link.next.next
self.pos.del_node(rem_node.link.next)
self.pos.del_node(rem_node.link)
new_first_link.val = abs(self.series[new_2nd_link.idx].val - self.series[
    new_first_link.idx].val)
self.heap_bubble_down(new_first_link.heap_idx)

F.2.7 PSO

import abc
import copy
import time
from Algos import sga as nlg
import random
import numpy as np
import Algos.optim_virtual as ov

class DomainError(Exception):
    def __str__(self):
        return "Domain size zero"

class individual:
    ""
    Class implementation of a single individual in PSO population
    ""

    Args:
    domain (list): indicates which indices of the time series this individual has access to
    master (PSO): PSO object whose population this individual is a part of
    this_idx (int): unique integer assigned to this individual in the population
    pos (list): If given, initial position (or solution) for this individual
    ""

    __slots__ = ["vel", "pos", "pers_best", "val", "dim", "this_idx", "pers_best_pos",
                "pers_best_val", "best_nbr_idx", "left_border", "right_border"]
def __init__(self, domain, dim, master, this_idx, pos=None):
    self.left_border, self.right_border = master.get_nbr_bounds(this_idx)
    self.dim = dim
    self.pos = [0, 0]
    if domain[1] - domain[0] == 0:
        raise DomainError
    gen_size = (domain[1] - domain[0] + 1)/dim
    if gen_size < 1:
        self.pos = list(range(domain[1] - domain[0] + 1))
        return
    if pos is None:
        while self.pos[0] == self.pos[1]:
            try:
                self.pos = \
                np.asarray([random.randrange(int(gen_size*i),\
                             int(gen_size*(i + 1)))\n                             for i in range(dim)])
            except ValueError:
                raise ValueError("gen_size:", gen_size, ", domain:", domain)
        else:
            self.pos = pos
            self.pos.sort()
            self.pos = np.asarray(self.pos).astype(np.float64)
    self.vel = np.random.random(size=self.dim)*domain[1]/self.dim/2\
               - domain[1]/self.dim/4

def init_pers_best(self):
    self.pers_best_pos = copy.deepcopy(self.pos)
    self.pers_best_val = -float("inf")

class PSO(Optim):

def __init__(self, dim, pop_size, nbr_rad, cog_scale_info,\
             soc_scale_info, inertia, max_time, mutate, trans_cost, \
             no_change_stop_ratio, max_iter, long=True, from_hierarch=False,\
             return_top=None, nbr_rad_rate=None, **kwargs):
    self.cog_scale_info = copy.deepcopy(cog_scale_info)
    self.cog_scale = copy.deepcopy(cog_scale_info)
    self.soc_scale_info = copy.deepcopy(soc_scale_info)
    self.soc_scale = copy.deepcopy(soc_scale_info)
self.inertia = max(0.05, abs(inertia))
self.max_time = max_time
self.pop_size = max(1, int(np.round(pop_size)))
if nbr_rad_rate is not None:
    nbr_rad = nbr_rad_rate*self.pop_size
self.nbr_rad = min(max(0, int(np.ceil(nbr_rad))), self.pop_size)
self.nbr_rad_rate = nbr_rad_rate
self.return_top = int(np.round(return_top))

self.no_change = 0
self.iter_count = 0

self.max_iter = max(0, int(np.round(max_iter)))
self.no_change_stop_ratio = max(0.005, no_change_stop_ratio)
self.no_change_stop = max(0,
    int(np.round(no_change_stop_ratio*self.max_iter)))

super().__init__(dim, long, trans_cost, from_hierarch)

if self.long:
    self.search_up = 1
else:
    self.search_up = -1

def __str__(self):
    return "PSO"

def init_pop(self, var_dist=None, new_dim=None, new_max_time=None):
    self.soc_scale = copy.deepcopy(self.soc_scale_info)
    self.cog_scale = copy.deepcopy(self.cog_scale_info)
    self.begin_time = time.perf_counter()
    if new_dim is not None:
        self.dim = new_dim
    self.pop = []
    if not self.data_set:
        print("No data available - population vector empty")
        return
    for i in range(self.pop_size):
        pos = None
        if var_dist is not None:
            pos = np.asarray(var_dist.sample(self.dim))
```python
self.pop.append(individual((0, len(self.data.reduced) - 1),
    self.dim, pos=pos, master=self, this_idx=i))
self.pop[-1].init_pers_best()

for i in range(self.pop_size):
    self.evaluate(i)

def update_velocity(self, cog_sc=None, soc_sc=None, indiv_idx=None):
    """Updates velocity

    if indiv_idx is None, then cognitive and social scale
    parameters are updated first (via the linear function, either from
    0.5 to 2.5 or from 2.5 to 0.5), then update_velocity is called for
    each individual. If indiv_idx is not None, the individual defined
    by the corresponding index has its velocity updated

    Args:
        cog_sc (float or None): cognitive scale parameter
        soc_sc (float or None): social scale parameter
        indiv_idx (int or None): index of individual which will have its velocity updated

    returns None
    ""
    if (not (indiv_idx is None)):
        if (self.nbr_rad > 0) and (self.nbr_rad < self.pop_size):
            social_pos, social_val, best_nbr_idx = self.get_best_nbr(indiv_idx)
            self.pop[indiv_idx].best_nbr_idx = best_nbr_idx
        else:
            social_pos = self.glob_best_pos
        self.pop[indiv_idx].vel = self.pop[indiv_idx].vel*self.inertia
            + np.multiply(np.asarray([random.random() for i in range(self.dim)])*cog_sc,
                (self.pop[indiv_idx].pers_best_pos
                - self.pop[indiv_idx].pos))
            + np.multiply(np.asarray([random.random() for i in range(self.dim)])*soc_sc,
                (social_pos -
                self.pop[indiv_idx].pos))
        for idx, v in enumerate(self.pop[indiv_idx].vel):
            if idx == 0 and self.pop[indiv_idx].pos[idx] + v < 0:
```
elif idx == self.dim - 1 and self.pop[indiv_idx].pos[idx] + v > len(self.data.reduced) - 1:
    self.pop[indiv_idx].vel[idx] = \n    len(self.data.reduced) - 1 - self.pop[indiv_idx].pos[idx]
elif (idx > 0) and (idx < self.dim - 1) and
    (self.pop[indiv_idx].pos[idx] + v >= self.pop[indiv_idx].pos[idx + 1]):
elif (idx > 0) and (idx < self.dim - 1) and
    (self.pop[indiv_idx].pos[idx] + v <= self.pop[indiv_idx].pos[idx - 1]):
return
else:
    cog_sc = self.cog_scale.new_scale()
    soc_sc = self.soc_scale.new_scale()
    for idx, obj in enumerate(self.pop):
        self.update_velocity(cog_sc, soc_sc, idx)
    return

def update_pos(self, indiv_idx = None):
    """Updates position

    If indiv_idx is None, then all individuals in the population have
    their positions updated. If indiv_idx is not None, then the
    individual specified by indiv_idx has its position updated

    Args:
    indiv_idx (int or None): index of individual to be updated, or None
        for whole population
    """
    if not (indiv_idx is None):
        self.pop[indiv_idx].pos += self.pop[indiv_idx].vel
    else:
        for idx in range(len(self.pop)):
            self.update_pos(idx)
        self.pop[idx].pos =
        np.asarray(list(map(self.fix_bounds,\
self.pop[0].pos))
    self.pop[0].pos.sort()
    self.evaluate(0)
    if self.pop[0].val > self.glob_best_val:
        self.glob_best_pos = copy.deepcopy(self.pop[0].pos)
        self.glob_best_val = self.pop[0].val

def fix_bounds(self, pos_val):
    if pos_val >= len(self.data.reduced):
        if len(self.data.reduced) >= 4:
            return len(self.data.reduced) - 1 - \n               random.randrange(int(len(self.data.reduced))//4)
        else:
            return len(self.data.reduced) - 1
    elif pos_val < 0:
        if len(self.data.reduced) >= 4:
            return random.randrange(int(len(self.data.reduced))//4)
        else:
            return 0
    elif pos_val >= len(self.data.reduced) - 1:
        return len(self.data.reduced) - 1
    else:
        return pos_val

def update_all(self):
    self.update_velocity()
    self.update_pos()

def _optimise(self, **kwargs):
    self.no_change = 0
    self.iter_count = 0
    self.glob_best_pos, self.glob_best_val = self.get_glob_best()
    prev_glob_best_val = self.glob_best_val
    self.glob_best_pos = np.asarray([p for p in self.glob_best_pos])
    if (self.no_change_stop <= 0) or (self.max_iter <= 0):
        if self.return_top is None or (self.return_top == 1):
            return self.glob_best_pos.astype(np.int64), self.glob_best_val
        else:
            return [(indiv.pers_best_pos.astype(np.int64),
                indiv.pers_best_val) for indiv in self.get_top_k()]
    else:
        return [(indiv.pers_best_pos.astype(np.int64),
                indiv.pers_best_val) for indiv in self.get_top_k()]

        condition_true = True
while condition_true:
    self.update_all()
    self.iter_count += 1

    if isinstance(self.max_time, float) or isinstance(self.max_time, int):
        condition_true = time.perf_counter() - self.begin_time < self.max_time
    if condition_true:
        condition_true = (self.no_change < self.no_change_stop) and\
        (self.iter_count < self.max_iter)

    if prev_glob_best_val == self.glob_best_val:
        self.no_change += 1
    if (self.return_top is None) or (self.return_top == 1):
        return self.glob_best_pos.astype(np.int64), self.glob_best_val
    else:
        return [(indiv.pers_best_pos.astype(np.int64), indiv.pers_best_val)\
            for indiv in self.get_top_k()]

def get_glob_best(self):
    glob_best_pos = self.pop[0].pers_best_pos
    glob_best_val = self.pop[0].pers_best_val
    for idx, particle in enumerate(self.pop[1:]):
        if particle.pers_best_val > glob_best_val:
            glob_best_val = particle.pers_best_val
            glob_best_pos = particle.pers_best_pos
    return copy.deepcopy(glob_best_pos), glob_best_val

def get_top_k(self):
    sorted_pop = sorted(self.pop,key=lambda x: x.pers_best_val, reverse=True)
    return sorted_pop[:self.return_top]

def get_nbr_bounds(self, idx):
    """Get the range of the neighbourhood for individual defined by idx

    The bound of the neighbourhood is defined by the
    index in the population vector of the first individual in the neighbourhood
    and the index of the last individual in the neighbourhood.

    returns (tuple): (starting index, end index)
    """
    if idx < self.nbr_rad:
begin = idx - self.nbr_rad + self.pop_size
else:
    begin = idx - self.nbr_rad

if idx > len(self.pop) - 1 - self.nbr_rad:
    end = idx - len(self.pop) + self.nbr_rad
else:
    end = idx + self.nbr_rad

return begin, end

def get_best_nbr(self, indiv_idx):
    """Returns best neighbour for specific individual
determines and returns the best neighbour for the individual
represented by indiv_idx.

Args:
    indiv_idx (int): index of individual for which neighbourhood best
        is to be determined
returns (tuple): (position, fitness, index) of best neighbour
""
    begin, end = self.get_nbr_bounds(indiv_idx)

    if indiv_idx == 0:
        whole_search = True
    else:
        if self.pop[indiv_idx - 1].best_nbr_idx ==
            self.pop[indiv_idx - 1].left_border:
            whole_search = True
        else:
            whole_search = False
    if whole_search:
        best_nbr_pos = self.pop[end].pers_best_pos
        best_nbr_val = self.pop[end].pers_best_val
        best_nbr_idx = end
        if begin > end:
            for idx in range(begin, self.pop_size):
                if self.pop[idx].pers_best_val > best_nbr_val:
                    best_nbr_pos = self.pop[idx].pers_best_pos
best_nbr_val = self.pop[idx].pers_best_val
best_nbr_idx = idx
for idx in range(end + 1):
    if self.pop[idx].pers_best_val > best_nbr_val:
        best_nbr_pos = self.pop[idx].pers_best_pos
        best_nbr_val = self.pop[idx].pers_best_val
        best_nbr_idx = idx
else:
    for idx in range(begin, end):
        if self.pop[idx].pers_best_val > best_nbr_val:
            best_nbr_pos = self.pop[idx].pers_best_pos
            best_nbr_val = self.pop[idx].pers_best_val
            best_nbr_idx = idx
return best_nbr_pos, best_nbr_val, best_nbr_idx
else:
    if self.pop[self.pop[indiv_idx - 1].best_nbr_idx].pers_best_val >
    self.pop[self.pop[indiv_idx].right_border].pers_best_val:
        return self.pop[self.pop[indiv_idx - 1].best_nbr_idx].pers_best_pos,
        self.pop[self.pop[indiv_idx - 1].best_nbr_idx].pers_best_val,
        self.pop[indiv_idx - 1].best_nbr_idx
    else:
        return self.pop[self.pop[indiv_idx].right_border].pers_best_pos,
        self.pop[self.pop[indiv_idx].right_border].pers_best_val,
        self.pop[indiv_idx].right_border

def evaluate(self, indiv_idx):
    indiv = self.pop[indiv_idx]
    total = 0.0
    ineq_count = 0
    for pos_idx in range(1, len(indiv.pos), 2):
        total += self.data.reduced[int(np.round(indiv.pos[pos_idx]))].val -
        self.data.reduced[int(np.round(indiv.pos[pos_idx - 1]))].val
    trans_cost = 0.0
    if self.trans_cost is not None:
        for pos_idx in range(1, len(indiv.pos), 2):
            trans_cost += self.trans_cost*(self.data.reduced[int(np.round(indiv.pos[
                pos_idx]))]).val
            + self.data.reduced[int(np.round(indiv.pos[
                pos_idx - 1]))].val)
    indiv.val = self.search_up*total
    indiv.val -= trans_cost
if indiv.val > indiv.pers_best_val:
    indiv.pers_best_val = indiv.val
    indiv.pers_best_pos = copy.deepcopy(indiv.pos)

---

**F.2.8 PSO-SGA**

class PSORefined(PSO):
    
    def __init__(self, nlg_pop_size, dim, pop_size, nbr_rad, cog_scale_info,\
                 soc_scale_info, inertia, max_time, mutate, trans_cost,\
                 return_top, no_change_stop_ratio, max_iter, long=False,\
                 from_hierarch=False, nbr_rad_rate=None, *args):
        self.nlg_pop_size = max(0, int(np.round(nlg_pop_size)))
        self.pso_obj = PSO(dim=dim, pop_size=pop_size, nbr_rad=nbr_rad,\
                           cog_scale_info=cog_scale_info,\
                           soc_scale_info=soc_scale_info, inertia=inertia,\
                           max_time=max_time, mutate=mutate, trans_cost=trans_cost,\
                           no_change_stop_ratio=no_change_stop_ratio,\
                           max_iter=max_iter, return_top=return_top,\
                           from_hierarch=from_hierarch, nbr_rad_rate=nbr_rad_rate,\
                           long=long, *args)
        ov.Optim.__init__(self, dim, long, trans_cost, from_hierarch)
        self.pop_size = self.pso_obj.pop_size
        self.nbr_rad = self.pso_obj.nbr_rad
        self.cog_scale_info = self.pso_obj.cog_scale_info
        self.soc_scale_info = self.pso_obj.soc_scale_info
        self.inertia = self.pso_obj.inertia
        self.max_time = self.pso_obj.max_time
        self.no_change_stop = self.pso_obj.no_change_stop
        self.max_iter = self.pso_obj.max_iter
        self.nbr_rad_rate = self.pso_obj.nbr_rad_rate
        self.return_top = self.pso_obj.return_top
        self.no_change_stop = max(1, self.no_change_stop)
        self.max_iter = max(1, self.max_iter)
    
    def __str__(self):
        return "PSO Refined"
    
    def get_params(self):
        
        ---
return {"pop_size": self.pop_size, "nbr_rad": self.nbr_rad,\
"inertia": self.inertia, "max_iter": self.max_iter,\
"no_change_stop": self.no_change_stop,\
"nlg_pop_size": self.nlg_pop_size, "return_top": self.return_top}

def init_pop(self, new_dim=None):
    if new_dim is not None:
        self.dim = new_dim
    self.pso_obj.init_pop(new_dim=new_dim)

def set_data(self, data):
    super().set_data(data)
    self.pso_obj.set_data(data)

def _optimise(self, **kwargs):
    best_list = self.pso_obj.optimise()
    self.iter_count = self.pso_obj.iter_count
    if not isinstance(best_list, list):
        best_list = [best_list]
    if self.nlg_pop_size <= 0:
        return best_list[0]
    highest_val = -float("inf")
    highest_pos = None
    self.raw_pso_fitness = best_list[0][1]
    for pos, val in best_list:
        nlg_inst = nlg.SGA(init_pos=pos, pop_size=self.nlg_pop_size,\
                             long=self.long, trans_cost=self.trans_cost)
        nlg_inst.set_data(self.data)
        nlg_inst.init_pop()
        new_pos, new_val = nlg_inst.optimise()
        if new_val > highest_val:
            highest_val = new_val
            highest_pos = new_pos
    return highest_pos, highest_val

F.2.9 S-Race

class SRace:
    def __init__(self, init_pool, delta, filepath, new_dim=None,\
file_count=None, is_race_iter=None):
    self.pool = create_pool(init_pool)
    self.init_pool_size = len(self.pool)
    self.delta = delta
    if isinstance(filepath, str):
        self.all_series = ud.make_series_basic(filepath, False)
    else:
        self.all_series = filepath
    self.result_array = np.zeros((1, len(self.pool), 2))
    self.dominance_matrix = np.zeros((len(self.pool), len(self.pool)))
    self.series_idx = np.random.choice(range(len(self.all_series)), replace=False, size=len(self.all_series))
    self.new_dim = new_dim
    self.file_count = file_count
    self.is_race_iter = is_race_iter
    self.out_idx = 0
    self.iteration_performs = [(None, None) for i in range(len(self.pool))]

def eval_all(self):
    """Evaluate all models

calling this function runs all models on the time series
    to be sampled for the current iteration.
    ""
    if not (np.sum(self.result_array) == 0):
        self.result_array = np.append(self.result_array,
                                       np.zeros((1, len(self.pool), 2)), 0)
    self.iteration_performs = [None for i in range(len(self.iteration_performs))]
    for (idx, optim_obj) in enumerate(self.pool):
        optim_obj[0].set_data(self.curr_series)
        begin_time = perf_counter()
        optim_obj[0].init_pop(new_dim=self.new_dim)
        pos, val = optim_obj[0].optimise()
        time_taken = perf_counter() - begin_time
        time_taken = max(1e-12, time_taken)
        if hasattr(optim_obj[0], "max_time"):  
            time_taken = optim_obj[0].max_time
        self.result_array[-1][idx][0] = val
        self.result_array[-1][idx][1] = 1/time_taken
        self.iteration_performs[optim_obj[1]] = (val, time_taken)
def det_dominance(self, pool_idx):
    """Update dominance matrix

    adjusts the dominance matrix based on performance of models in
    last iteration

    Args:
        pool_idx (int): index of model against which all other models are
                       compared
    returns None
    """
    for i in range(len(self.pool)):
        if not (i == pool_idx):
            diff_vec = np.asarray(self.result_array[-1][pool_idx]) -
                        np.asarray(self.result_array[-1][i])
            diff_vec = [el for el in diff_vec
                         if not math.isclose(el, 0.0, abs_tol=1e-6)]
            logical_vec = list(map(lambda x: x > 0, diff_vec))
            if (len(logical_vec) > 0) and all(logical_vec):
                self.dominance_matrix[pool_idx, i] += 1
    self.result_array = np.asarray(self.result_array)

def compare(self, comp_idx):
    """Compare all models against a single model

    compares different models to obtain p-values, after which
    holm-bonferroni procedure is called

    Args:
        comp_idx (int): index of model to be compared against all
                        other models
    returns (list): list of rejected models
    """
    comps = []
    for i in range(len(self.pool)):
        if not (i == comp_idx):
            if self.dominance_matrix[comp_idx, i] >
                self.dominance_matrix[i, comp_idx]:
                n = int(self.dominance_matrix[comp_idx, i] +
                        self.dominance_matrix[i, comp_idx])
pi_sig = custom_binom(n=n, p=0.5, \
    k=int(self.dominance_matrix[comp_idx, i]))
comps.append((pi_sig, (comp_idx, i)))
if len(comps) == 0:
    return set()
comps.sort(key=lambda x: x[0])
return self.holm(comps)

def holm(self, lst):
    '''Holm-bonferroni procedure.'''
    Args:
    lst (3-d array): information on the p-values for each of the comparisons
    made
    returns (list): list of rejected models
    '''
    rejects = set([lst[i][1][1] for i in range(len(lst))])
    for k in range(1, len(lst) + 1, 1):
        if lst[k - 1][0] > self.alpha/(len(lst) + 1 - k):
            rejects = set([lst[i][1][1] for i in range(k-1)])
            break
    return rejects

def remove_rejects(self, rejects):
    rejects = sorted(rejects, reverse=True)
    for rej in rejects:
        del self.pool[rej]
        self.dominance_matrix = np.delete(self.dominance_matrix, rej, 0)
        self.dominance_matrix = np.delete(self.dominance_matrix, rej, 1)
        self.result_array = np.delete(self.result_array, rej, 1)

def iteration(self, choice_idx):
    '''A single S-Race iteration

    when this function is called, a single S-Race iteration
    gets executed.
    '''
    Args
    choice_idx (int): index of the time series to do the iteration on
    returns None
```python
self.alpha = (1 - self.delta)/(self.max_iters*(self.init_pool_size - 1))
self.curr_series = self.all_series[choice_idx]
self.eval_all()
pool_idx = 0
checked = []
while not self.all_checked(checked):
    for pool_idx in range(len(self.pool)):
        if self.pool[pool_idx][1] not in checked:
            break
    checked.append(self.pool[pool_idx][1])
    self.det_dominance(pool_idx)
    if (self.result_array.shape[0] > 5):
        rejects = self.compare(pool_idx)
        self.remove_rejects(rejects)

def all_checked(self, checked):
    return all([el[1] in checked for el in self.pool])

def run(self, iters):
    """Run S-Race

    Args:
    iters (int): number of iterations for which S-Race is to be run
    returns (list<model>): returns the list of models that survived S-Race
    ""
    self.max_iters = iters
    self.curr_iter = 0
    count = 0
    series_idxes = list(range(iters))
    series_idxes = [idx % 100 for idx in series_idxes]
    while (count < len(series_idxes)) and (len(self.pool) > 1):
        if len(self.pool) == 2:
            logical_vec = [isinstance(optim_obj, BU) or
                            isinstance(optim_obj, SGA)
                           for optim_obj, idx in self.pool]
            if logical_vec[0] and logical_vec[1]:
                break
        self.iteration(count % 100)
        count += 1
        self.curr_iter = count
```
return self.pool